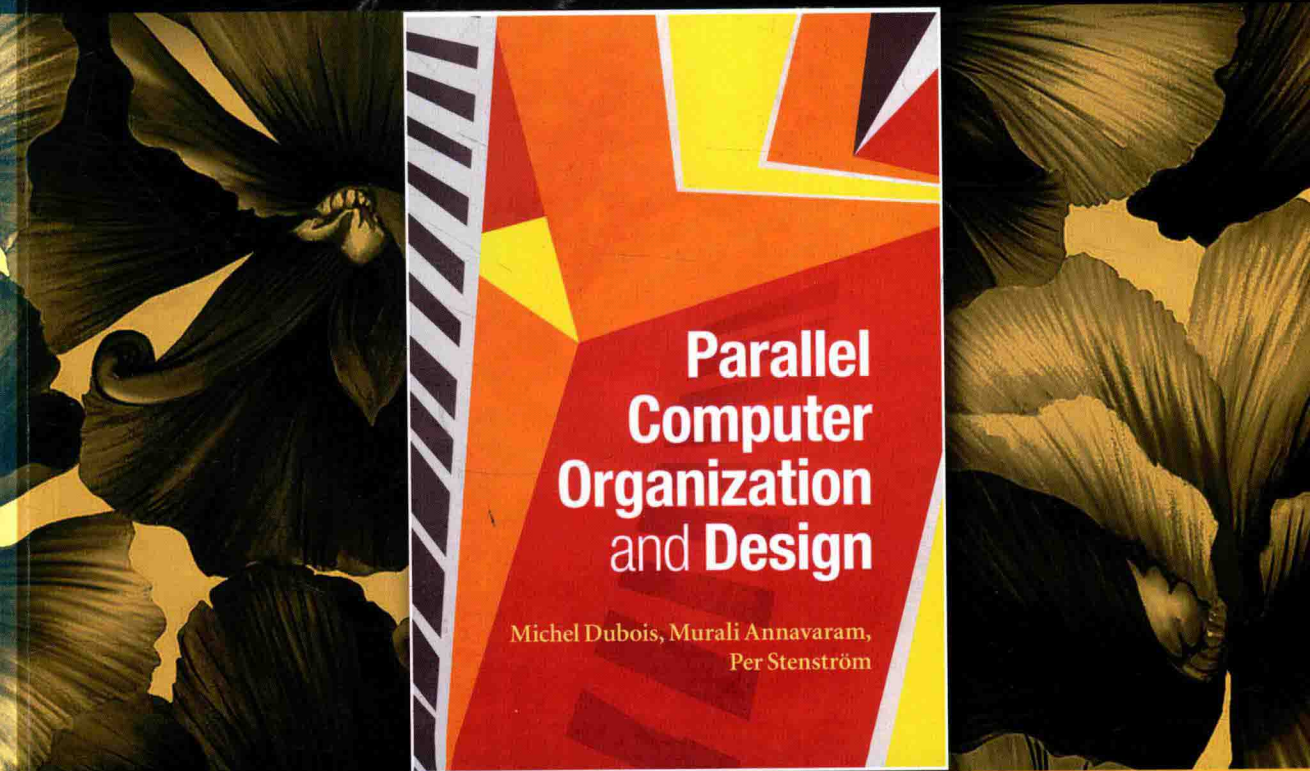


并行计算机组成 与设计

[美] 米歇尔·杜波依斯 (Michel Dubois)
穆拉里·安纳瓦拉姆 (Murali Annavaram) 著
[瑞典] 佩尔·斯坦斯托姆 (Per Stenström)
范东睿 叶笑春 王达 译

Parallel Computer Organization and Design



计 算 机 科 学 丛 书

并行计算机组成 与设计

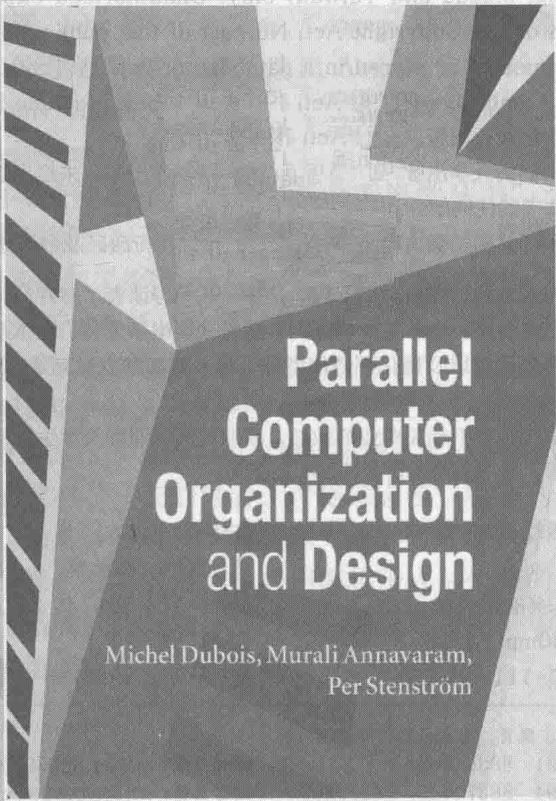
[美] 米歇尔·杜波依斯 (Michel Dubois)

穆拉里·安纳瓦拉姆 (Murali Annavaram) 著

[瑞典] 佩尔·斯坦斯托姆 (Per Stenström)

范东睿 叶笑春 王达 译

Parallel Computer Organization and Design



Parallel
Computer
Organization
and Design

Michel Dubois, Murali Annavaram,
Per Stenström



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

并行计算机组成与设计 / (美) 米歇尔·杜波依斯 (Michel Dubois) 等著; 范东睿, 叶笑春, 王达译. —北京: 机械工业出版社, 2017.3

(计算机科学丛书)

书名原文: Parallel Computer Organization and Design

ISBN 978-7-111-56223-8

I. 并… II. ①米… ②范… ③叶… ④王… III. 并行计算机—计算机体系结构 IV. TP338.6

中国版本图书馆 CIP 数据核字 (2017) 第 040466 号

本书版权登记号: 图字: 01-2014-1082

This is a Chinese simplified edition of the following title published by Cambridge University Press:

Michel Dubois, Murali Annavaram, Per Stenström, Parallel Computer Organization and Design, ISBN 978-0-521-88675-8.

© Cambridge University Press 2012.

This Chinese simplified edition for the People's Republic of China (excluding Hong Kong, Macau and Taiwan) is published by arrangement with the Press Syndicate of the University of Cambridge, Cambridge, United Kingdom.

© Cambridge University Press and China Machine Press in 2017.

This Chinese simplified edition is authorized for sale in the People's Republic of China (excluding Hong Kong, Macau and Taiwan) only. Unauthorized export of this simplified Chinese is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of Cambridge University Press and China Machine Press.

本书原版由剑桥大学出版社出版。

本书简体字中文版由剑桥大学出版社与机械工业出版社合作出版。未经出版者预先书面许可, 不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 销售。

本书以简单易懂的方式讲解错综复杂的并行体系结构, 引导读者了解并行计算机的工作原理, 同时鼓励读者创新并实现自己的设计。全书共 9 章, 内容涵盖底层电子工艺、微体系结构、存储结构、互连网络、多处理器、片上多处理器以及量化评估模型等。每一章都独立且完备, 既包含全面的基本概念, 也涵盖一些前沿研究点。

本书适合作为高等院校计算机相关专业的教材, 教师可根据课程及学生的层次选取不同的主题。同时, 对于工程师和研究者, 本书也是不可多得的有益参考。

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 曲 熠

责任校对: 李秋荣

印 刷: 三河市宏图印务有限公司

版 次: 2017 年 4 月第 1 版第 1 次印刷

开 本: 185mm × 260mm 1/16

印 张: 24

书 号: ISBN 978-7-111-56223-8

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来,源远流长的科学精神和逐步形成的学术规范,使西方国家在自然科学的各个领域取得了垄断性的优势;也正是这样的优势,使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中,美国的产业界与教育界越来越紧密地结合,计算机学科中的许多泰山北斗同时身处科研和教学的最前线,由此而产生的经典科学著作,不仅擘划了研究的范畴,还揭示了学术的源变,既遵循学术规范,又自有学者个性,其价值并不会因年月的流逝而减退。

近年,在全球信息化大潮的推动下,我国的计算机产业发展迅猛,对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇,也是挑战;而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下,美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此,引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用,也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始,我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力,我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系,从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson 等大师名家的一批经典作品,以“计算机科学丛书”为总称出版,供读者学习、研究及珍藏。大理石纹理的封面,也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助,国内的专家不仅提供了中肯的选题指导,还不辞劳苦地担任了翻译和审校的工作;而原书的作者也相当关注其作品在中国的传播,有的还专门为其书的中译本作序。迄今,“计算机科学丛书”已经出版了近两百个品种,这些书籍在读者中树立了良好的口碑,并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化,教育界对国外计算机教材的需求和应用都将步入一个新的阶段,我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

华章网站: www.hzbook.com

电子邮件: hzjsj@hzbook.com

联系电话: (010) 88379604

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037



华章科技图书出版中心

由于单核的性能发展遇到瓶颈，因此并行计算机和多核体系结构的重要性越来越显著。本书以通俗易懂的方式描绘了错综复杂的并行体系结构，这对学术研究人员和工程师都十分有用。

Shubu Mukherjee, Cavium 公司

这本书不仅可以帮助你清晰理解并行系统的原理，而且对于并行系统设计者来说也是不可多得的好书。

陈云霄，中国科学院计算技术研究所

未来的电子系统都会由内置微处理器组成，因此对体系结构的理解至关重要。从处理器的基本知识到内存结构再到片上多处理器，本书系统介绍了计算机体系结构的基本问题，通篇流畅，易于理解，是一本出色的工具书。

Uri Weiser, 以色列理工学院

这本书对于理解基本的片上技术、互连网络的限制以及它们对于计算机体系结构设计的影响很有帮助。所有计算机系统相关领域的学生和开发人员都应该首先掌握平台无关的单核和多核基本原理，正如本书所呈现的一样。

Mateo Valero, 巴塞罗那超算中心

在过去的几年里，处理器体系结构发生了由单核向多核的重大转变，这个领域中需要一本易懂且与时俱进的书。本书既可以作为了解多核和并行体系结构的入门书，也可以作为工程师和研究人员的参考书。

Olivier Temam, 法国 INRIA

高级计算机体系结构领域需要一本综合性强、权威性高但又浅显易懂的教科书和参考书，本书满足了这一需求。威斯康星大学三门关于计算机体系结构的课程所讲授的所有关键原理和概念，在本书中都得以呈现，并且组织良好、深思熟虑、易于教学，并没有用晦涩抽象的技巧以及大量的量化数据使得读者手足无措。特别是第8章讲述的片上多处理器的内容紧跟工业发展前沿，最后一章的量化评估更是点睛之笔，这些是本书能从同类书中脱颖而出的关键。

Mikko Lipasti, 威斯康星大学麦迪逊分校

本书对于计算机系统的各个方面都有深刻的见解，是本领域中一本综合性强、系统性完善并且紧跟时代发展的书。本书经过精心编排，保证每章都独立、完备，使读者无需阅读整本书就可以理解每章的主题。本书内容丰富、连贯、清晰，书中的问题也利于培养创造性思维。我推荐此书作为所有计算机设计领域的研究生、青年研究者和工程师

的必读书。

张立新，中国科学院计算技术研究所

并行体系结构是计算机系统获得高性能和高效率的关键。本书从多个层面讲述了并行体系结构的知识，包括从简单的晶体管到完全成熟的 CMP 等内容，学习本书的过程将是一段难忘的旅程。

Ronny Ronen, Intel 公司

多核芯片使得并行体系结构无处不在，也使得对并行体系结构的理解变得非常必要。本书系统地讲解了并行系统体系结构，并以简洁的方式介绍了 cache 一致性和内存一致性的基本原理，是用于研究生一学期教学的理想教科书。

Lawrence Rauchwerger, 得克萨斯农工大学

本书是现在并行体系结构方向最好的一本书，我计划在课上使用这本书，它以清晰明了的方式讲述了并行计算最前沿的知识。

Trevor Mudge, 密歇根大学

“并行”以不同的形式存在于多个层次，是未来计算机系统必不可少的，也是新一代计算机科学家和工程师必须掌握的。为了充分理解其中成百上千个不同选项之间的复杂关系，我们必须对其进行总结、排序和综合，这也正是本书所完成的杰出工作。我尤其想强调的是，三位作者用极其清晰的方式解释了一致性、同步和内存一致性等概念。

Manolis Katevenis, 希腊克里特大学

本书由业界顶尖专家所著，对并行计算机进行了全面讲解。在技术上严谨完备，同时也浅显易懂，还包括一些容易忽略但很重要的主题，如可靠性、功耗和模拟。

Norm Jouppi, 惠普公司

本书汲取了传统计算机体系结构的精华，也包括多核和并行系统方面的基本原理。本领域亟需这样一本高质量的书，作者很好地对主题进行了组织和改进，为下一代计算机架构师提供了设计多核和众核系统所需的基本原理。

David Kaeli, 美国东北大学

本书是该领域期盼已久的一本出色的教学用书，它将是计算机体系结构专业的学生和教师不可或缺的资源。

Josep Torrellas, 伊利诺伊大学

在 21 世纪初期，由于单核处理器的性能提升越来越困难，因此基于多核处理器的并行体系结构获得了快速发展，计算机体系结构的研究重心也开始转向并行体系结构，这给计算机体系结构的课程设计和教材编写都带来了很大挑战。

我们从 2005 年开始从事多核和众核体系结构的研究，一直在寻找一本系统讲解并行体系结构的优秀入门教材和参考书，本书的出现令我们眼前一亮，这正是我们一直想要的那本教材。

本书的作者是并行计算机体系结构领域的知名学者，教材内容的权威性无需多言。在前言中，作者也给出了非常详细的章节纲要及学习指导，建议读者参考。此外，本书之所以是一本极其出色的教材，我们认为原因还在于其具有以下特色：

内容完备且易懂。本书在重点介绍最新的并行计算机体系结构的同时，对传统单核体系结构中的基本概念和原理也进行了简要介绍，并且讲解方式浅显易懂，即使计算机体系结构知识背景比较薄弱的读者也可以较快入门。

每一章自成体系，内容相对独立且完备。研究人员和工程人员可以有针对性地快速切入相关章节进行学习和查阅，教师在教学环节中也可以根据学生实际情况选择相关章节施教，比如难度较大的第 7 章更适合研究生课程。

针对体系结构量化评估工具编写了专门的章节。模拟器是计算机体系结构研究和教学中极其重要的工具，然而现有的体系结构教材中往往忽略了对这类工具的介绍。本书在第 9 章中对量化评估工具进行了系统全面的阐述，这对体系结构领域的初学者来讲是非常有价值的。

本书的翻译工作由范东睿、叶笑春、王达共同协作完成。在翻译过程中，我们也修正了原书中的一些小错误。不过由于我们的水平和精力有限，可能会引入新的错误，恳请大家不吝赐教。

本书的完成得益于中科院计算所许多老师和同学的无私支持。衷心感谢张志敏研究员对译稿部分章节的审校，感谢李文明、吴萌、向陶然、严明玉、李戈、张伍召、张承龙等对译稿的贡献。另外，本书的出版还得到了机械工业出版社的大力帮助，在此也对出版社各位同仁在翻译和审校等环节的辛勤付出表示感谢。

译者

2017 年 2 月

在飞速发展的技术驱动下，计算机体系结构成为一个快速发展的领域。自 20 世纪 90 年代中期以来，计算系统的速度和可靠性显著增强，这主要得益于技术的发展、更快的主频和更深的流水线。这些改进将高性能计算机带给大众，对社会产生了深远影响，促进了网络创新和人类活动中生产力的大幅提升。我们所处的信息革命如同 18 世纪的工业革命，没有人会否认这次革命得益于技术的发展和微处理器体系结构的发展。

但是，这样快速的发展在未来可能无法维持下去，流水线深度已经达到可用的极限，由于功耗限制，主频也无法大幅突破。随着技术的发展以及片上资源的变少，可靠性、复杂性和功耗成为计算机设计考虑的首要问题，而不再是传统上考虑的成本、面积和性能。这些趋势促进了并行处理和并行体系结构的发展，因为这是解决当前和未来可能面临的体系结构问题的一条新的甚至可能是唯一的途径。人们普遍认为，我们需要利用并行处理才能使计算机领域呈现一片新的景象，而这个巨大的改变会产生深远的社会影响。因此，无论是工业界还是学术界，对并行体系结构的兴趣都已从工程上的好奇转换为实在的任务。

随着时间的流逝，各层次的并行化已经成为现代计算机系统发展的瓶颈。多处理器结构通过连接多个处理器提供了可扩展的性能表现，并已在高端系统领域称霸数十载。多处理器开发线程级并行（TLP），允许大型应用拥有很多线程，如计算机图形、科学/工程计算、数据库管理以及通信服务。随着体系结构和编译器技术的发展，微体系结构则开发指令级并行（ILP），并且获得了良好的性能表现。内存系统结构为了跟上指令吞吐量的需求，通过允许同时访问大量数据并保证执行的正确性而获得了快速发展。互连和相关的协议也不断改进，可以有效连接成百上千个处理器以及主频为几 GHz 的芯片。最近，微处理器的体系结构集成了系统级并行结构的范例，如向量处理和多处理器。在片上多处理器时代，每个微处理器都有多个核或 CPU，每个核可以并发执行多个线程。

并行体系结构很难设计也很难编程，我们必须理解并行体系结构带来的问题。本书针对最新的指令级并行和线程级并行技术给出了清晰易懂的讲解，此外，还将可靠性和功耗作为设计目标进行讲解。先前计算机体系结构方面的教材主要将性能作为设计考虑的核心问题。然而，现在尽管性能依旧是设计中的一个主要问题，但是复杂性、功耗和可靠性等其他问题也成为很重要的设计因素，这本关于并行计算机体系结构的新书将会讲解这些内容。

本书的基本目的是解释并行体系结构如何工作以及分析当今并行体系结构的正确设计，尤其是在技术受限的情况下。我们一般不会给出性能数据，并且尽力回避系统的具体细节描述。我们鼓励读者阅读发布在相关会议或期刊上的资料，详细的参考书目和历史发展回顾将会发布在网上。这会留下更多空间来讲解设计的基本问题，同时鼓励学生思考、创新、实现自己的设计。为了达到实践和创新的层次，全面了解现存的设计和实际问题以及限制因素是必需的过程。本书利用丰富的例子来解释概念，并且激发读者自己思考。此外，本书还用两章（第 8 章和第 9 章）的篇幅描述了一些工业界和学术界使用的系统和工具。

习题是学习的重要部分，每章之后的习题都采用问答题形式，有些很长很难的题目被

分成了多个子问题。习题的主要目的是给学生创造机会以对每章的概念有深刻的理解，并且培养学生的抽象思维能力。

本书适合对计算机体系结构感兴趣的计算机、电子工程和计算机科学专业的高年级本科生以及研究生阅读，也适合计算机行业中的工程师参考。由于本书涵盖了从微处理器到多处理器的大量知识，既包括基本内容也包含一些前沿的研究点，因此在教学时，可以通过选择不同的章节来适应不同的难度级别。学生可以学到硬件结构和组成多处理器的各个部分，以及技术发展趋势对于体系结构的影响，还有与性能、功耗、可靠性和功能正确性相关的设计问题等。本书可以用于研究生的基础课程，也可以用于接下来的高阶研究课程。本书的预修课程是计算机体系结构及组成，涵盖指令集和简单流水线处理器体系结构等内容。例如五级流水线及其控制相关的知识，包括前递、停顿和刷新等问题，这些可以帮助学生理解微处理器章节中更复杂的硬件问题。为了体现完整性，本书也包括了基本的指令集、流水线和存储等相关概念。由于现代微体系结构会影响多处理器的特征，因此了解其工作原理是必需的，此外还需要一定的编程知识。

内容纲要

本书整体内容完备，同时我们也尽力保证每一章都是独立且完备的，这可能会导致一些重复。本书共9章。第1章给出了计算机体系结构领域的基本观点，概述处理器、内存、互连网络、性能问题（主要是如何评判计算机系统）以及讨论技术发展对未来体系结构的影响。

对工艺实现细节的理解也是非常重要的，因为很多体系结构的设计决策都受底层工艺的影响。因此在第2章中，我们对CMOS进行回顾并探讨相关问题。有VLSI设计背景的学生可以略过某些讲解。这一章主要是针对计算机科学专业的学生，他们可能对电子工程和CMOS技术只有粗略的了解。这部分知识对理解本书其他部分没有影响，但是可以帮助学生理解为什么如今的体系结构会是这样，并了解一些设计决策的原因。这一章同其他章节的明显区别在于只介绍工艺相关的知识。

第3、4和6章讲述了并行系统设计的几个基本方面：处理器、存储和互连网络。第3章讲述微体系结构，对指令集和基本的机器结构（如五级流水线）进行了概括介绍，并在这一过程中给出了本书其他部分用到的指令集和基本ISA原理。这一章重点强调异常以及异常处理方法，因为异常对于开发微体系结构中的并行性有很大影响。有体系结构背景的学生可以略过该章的大部分内容。该章主要介绍通过各种软硬件方法来开发指令级并行。首先，考虑静态调度处理器（包括超标量处理器，它是五级流水线的扩展）中的设计问题。由于没有哪种机制可以进一步优化静态调度的指令并开发ILP，因此编译器技术就显得十分重要。而动态乱序（OoO）执行处理器则可以在几百条指令的执行窗口内重新动态调度（编译后）指令。在第3章，我们会一步步讲解OoO处理器的发展，从Tomasulo算法开始一直到可以推测调度的推测执行处理器，这也是目前最先进的OoO处理器。乱序执行处理器是处理器体系结构设计的一个极端，由于其支持动态调度，因而随着并行指令数的增加会带来复杂性和功耗等问题。并行微体系结构设计的另一个极端是超长指令字（VLIW）结构。在这种结构中，在编译阶段就决定了所有的动作（包括什么时候取指、译码和开始执行），这极大地降低了硬件复杂性和功耗。而实际处理器设计可以在两个极端之间进行折中，显式并行指令计算（Explicitly Parallel Instruction Computing, EPIC）就是这

样的一个尝试。最后，处理器还通过向量微体系结构来开发细粒度并行，向量处理在性能和功耗方面都很有效，并且广泛用于多媒体和信号处理应用中。

第4章讲解存储层次结构在硬件层的特点。如今，需要多层次并发存储结构来为并行微体系结构提供足够的指令和数据。这包括非阻塞 cache 设计以及软硬件预取策略。这些方法必须保证内存行为的正确性。另一个理解并行体系结构的重要因素是虚拟存储系统。正是因为有了虚存，现代体系结构必须能处理精确异常，多处理器必须采取相关的机制来保证每个处理器虚存的一致性（在第5章中介绍）。

第6章主要讲解互连网络问题。互连网络连接系统部件（系统区域网络，SAN）或者片上资源（片上网络，OCN）。由于并行体系结构性能的提高需要允许多个部件的并行访问，因此互连网络的设计对性能和功耗都极其重要，设计空间很大。第6章提供了互连网络设计的完整介绍，包括性能模型、交换策略、网络拓扑、路由算法和交换结构等。

第5、7和8章是关于多处理器的。第5章给出了消息传递和共享内存多处理器的基本结构和机制。首先通过程序例子讲解编程模型和基本的应用编程接口，这样可以方便读者理解体系结构中需要的各种机制。该章分层给出了消息传递所需的基本体系结构支持，从各种消息传递原语到实现所需的基本交换协议，再到加速所需的相关硬件支持。第5章其余部分关注共享内存系统。共享内存系统有几种可行的组织方式，不过出于商业上的考虑，目前最主要的方式是利用现有的商业微处理器来搭建新的多处理器系统，这些现有的微处理器都有自己的 cache。共享内存系统的每个处理器和片上多处理器的每个核都有自己的私有指令 cache、数据 cache 和虚地址转换 cache。因此必须采取某些机制保证它们在结构上的一致性。该章将介绍总线系统和分布式共享内存系统（cc-NUMA 和 COMA）这两大类结构。

第5章介绍了共享内存系统的体系结构，而第7章则主要解决共享内存多处理器的逻辑属性问题，包括同步、一致性和存储一致性模型，这三者之间有着紧密且微妙的联系。同步原语和对应的机制对于多线程程序的执行至关重要，必须要有硬件支持。在多个 cache 和缓存中同一个地址对应的多个备份需要保证一致性。共享内存系统最终的正确性在于存储一致性，它规定了访存的动态交错顺序。本书从静态调度处理器和动态调度处理器两方面描述了存储一致性模型的实现。第7章是本书中理论性最强的一章，但是也并不要求读者具备相关的理论背景。

第8章讲解片上多处理器（CMP）。由于具有集成密度高、通信延迟低的特点，因此 CMP 有可能带来新的、更简单高效的编程模型。在 CMP 环境中，CPU 相对低廉，可以用于各种新型计算模式。本章包括 CMP 相关的诸多内容，如 CMP 体系结构、核内多线程、事务内存、推测多线程和辅助执行等。

最后，第9章关注计算机体系结构设计的量化评估模型。计算机体系结构的很多设计决策是基于一系列在面积、性能、功耗和可靠性等方面的权衡而得出的。任何旨在改进现有实现的设计都必须经过严格的量化评估以评价这些改进的有效性。因此，学生和实践人员需要理解探索设计空间的量化方法。在本章中，我们会介绍很多这类话题，比如模拟方法、采样技术、工作负载特征刻画方法等。

致谢

编写本书花费了5年时间，这是我们的本科生和研究生教学经验以及多年实践经验的

结晶。并行计算机体系结构和并行处理无论在短期还是长期都扮演着重要的角色。因此，我们将在并行体系结构和并行编程方面继续努力教授计算机科学和计算机工程专业的学生。本书是我们对这一目标的贡献。

Michel Dubois

多年来，我有幸受到很多杰出同事的影响，尽管很多人我并不相识。感谢普渡大学的导师 Fayé Briggs，在最需要的时候，他给了我研究的信心。感谢明尼苏达大学和普渡大学为我提供了面对世界所必需的研究生教育，这是无价的。

作为一名南加州大学的教师，我从我的博士研究生那里学到了很多，希望他们也能从我这里学到知识。学生们帮我建立和发展了我对于计算机体系结构的想法（有些是好的，也有些是坏的）和观点，他们是：Christoph Scheurich、Aydin Uresin、Jin-Chin Wang、Fong Pong、Luiz Barroso、Kangwoo Lee、Adrian Moga、Xiaogang Qiu、Jaehoon Jeong、Jianwei Chen 以及 Jinho Suh。尤其是 Jinho 在编写本书时给了我们很大的帮助。

当然，还要感谢我的父母 Solange 和 André，是他们把我带到这个世界，并且在我到美国继续研究生学习时依然支持我，虽然我的决定伤了他们的心。还要感谢妻子 Lorraine，感谢她对我工作的支持。

最后，我代表三位作者对剑桥大学出版社的团队表示感谢：Julie Lancashire，感谢她从一开始对本项目的热情；Sarah Matthews，她参与了大部分出版工作，使得我们在整个过程中都保持热情；Irene Pizzie，他帮助我们修改了语法错误，尽力确保了本书内容的一致性。

Murali Annavaram

当我接受 Michel Dubois 教授的邀请编写本书时，我完全不知道这部杰作将耗费的时间和精力。但是，经过两年的努力完成这本书后，我看到我们共同努力的结果比我们任何一个人独立完成的书都要好，这让我很满足。我希望通过这本书分享我多年来在工业和学术上的经验，书中的内容来自于我从这个领域的大师们那里学到的东西。尤其是 Edward S. Davidson 教授对我的影响最为巨大，我至今还怀念他的红色和蓝色墨迹。Yale Part 教授教会我如何教学，他的授课方式很有感染力。我还要感谢 Walid Najjar 教授以及 Farnam Jahanian 教授给我这个机会。

在我的职业生涯中，没有人的影响力可以超过 John Shen，他真的是一个非常棒的老板。我还要感谢以下这些人：Ed Grochowski，他是一个创造性思维大师；Bryan Black，他和他的 Austin 团队开阔了我的视野；Quinn Jacobson，他给了我足够的信任；Viji Srinivasan，我和他就预取机制进行了长时间的讨论。

我在南加州大学的研究是本书一些内容的基础。感谢美国国家科学基金（NSF）和诺基亚公司给我的研究资助，同时，我的研究工作也离不开我的研究生团队的帮助，他们是我撰写章节的试读者，并给出了习题的答案。这里尤其要感谢 Jinho Suh，他的能力非常全面，从统计到排版编辑都从未让我失望过。

Bob 和 Nancy 教会我回馈社会。Kirti、Kalpesh 以及密歇根大学和加州州立大学的团队给了我很大支持。在这里我无法把名字一一列全，感谢你们。

最后，感谢为我牺牲了个人时间的家庭成员们，谢谢你们！

Per Stenström

我对这本书的贡献很大程度上来源于多年来对并行计算机体系结构的研究。如果没有我的导师 Lars Philipson, 我无法完成这些, 他在 20 世纪 80 年代早期就独具慧眼地发现了共享内存多处理器技术的潜力。更重要的是, 他给了我科学正确的认识和完成研究的信心。我的博士研究生给了我灵感, 他们教会了我很多东西。我多年的研究成果都浓缩在这本书中, 我从我的博士生身上学到了很多, 感谢他们: Mats Brorsson、Fredrik Dahlgren、Magnus Ekman、Håkan Grahm、Mafijul Islam、Thomas Lundqvist、Magnus Karlsson、Jim Nilsson、Ashley Saulsbury、Jonas Skeppstedt、Martin Thuresson、M. M. Waliullah 以及 Fredrik Warg。

此外, 我还想感谢计算机体系结构社区的很多人, 他们对我的职业发展有很大的影响。对卡内基-梅隆大学、斯坦福大学、南加州大学以及 Sun 公司的访问, 很大程度上影响了我对计算机体系结构到底是什么的理解。我的灵感的另一个重要来源是与欧洲同行们在 HiPEAC 上的交流, 我希望可以列出所有我想感谢的人, 但是人数确实太多了, 谢谢你们所有人!

我们早先已经在查尔姆斯理工大学对本书进行了课堂测试, 得到了很多有用的反馈。我尤其要感谢下面这些给我们反馈的人: Bhavishya Goel、Ben Juurlink、Johan Karlsson、Sally McKee、Filippo Del Tedesco、Andras Vajda 以及 M. M. Waliullah。

最后, 也是最重要的, 感谢妻子 Carina 和女儿 Sofia 对我的无私支持 and 理解。

目 录

Parallel Computer Organization and Design

出版者的话

赞誉

译者序

前言

第1章 总述 1

1.1 什么是计算机体系结构 2

1.2 并行体系结构的基本组成 3

1.2.1 处理器 4

1.2.2 存储 6

1.2.3 互连 9

1.3 并行体系结构 10

1.3.1 指令级并行 10

1.3.2 线程级并行 10

1.3.3 向量和阵列处理器 11

1.4 性能 12

1.4.1 基准测试集 13

1.4.2 Amdahl 定律 15

1.5 技术挑战 19

1.5.1 功耗和能量 19

1.5.2 可靠性 19

1.5.3 连线延迟 20

1.5.4 设计复杂度 20

1.5.5 尺寸缩小极限和 CMOS 终点 21

习题 22

第2章 工艺及其影响 25

2.1 概述 25

2.2 电学基本定律 26

2.2.1 欧姆定律 26

2.2.2 电阻 26

2.2.3 电容 27

2.3 MOSFET 晶体管和 CMOS

反相器 27

2.4 工艺变更 30

2.5 功耗和能耗 31

2.5.1 动态功耗 31

2.5.2 静态功耗 35

2.5.3 功耗和能量指标 37

2.6 可靠性 38

2.6.1 故障和错误 38

2.6.2 可靠性指标 39

2.6.3 故障率和老化 40

2.6.4 瞬时故障 42

2.6.5 间歇性故障 44

2.6.6 永久性故障 48

2.6.7 工艺偏差及其对故障的

影响 48

习题 49

第3章 处理器微结构 51

3.1 概述 51

3.2 指令集架构 52

3.2.1 指令类型和操作码 53

3.2.2 指令混合 55

3.2.3 指令操作数 55

3.2.4 异常、陷阱和中断 58

3.2.5 存储一致性模型 60

3.2.6 本书的核心 ISA 60

3.2.7 CISC 和 RISC 61

3.3 静态调度流水线 63

3.3.1 经典五级流水线 64

3.3.2 指令乱序完成 69

3.3.3 超流水和超标量 CPU 72

3.3.4 分支预测 73

3.3.5 静态指令调度 73

3.3.6 静态流水线的优缺点 77

3.4 动态调度流水线 78

5.4.6 通信事件的分类	188	7.2.1 共享内存通信模型	239
5.4.7 TLB 一致性	190	7.2.2 硬件组件	241
5.5 可扩展共享内存系统	192	7.3 一致性和 store 原子性	244
5.5.1 目录协议的基本概念和术语	193	7.3.1 多处理器一致性的实现困难	244
5.5.2 目录协议实现方法	193	7.3.2 cache 协议	246
5.5.3 目录协议的扩展性	197	7.3.3 store 原子性	249
5.5.4 层次化系统	200	7.3.4 纯一致性	254
5.5.5 页面迁移和复制	201	7.3.5 store 原子性和访存交错	262
5.6 全 cache 共享内存系统	204	7.4 顺序一致性	262
5.6.1 基本概念、硬件结构和协议	204	7.4.1 顺序一致性的形式化模型	263
5.6.2 平坦 COMA	206	7.4.2 顺序一致性的访存顺序规则	265
习题	207	7.4.3 入站消息管理	266
第 6 章 互连网络	214	7.4.4 store 同步性	270
6.1 概述	214	7.5 同步	272
6.2 互连网络的设计空间	215	7.5.1 基本同步原语	273
6.2.1 设计概念综述	215	7.5.2 基于硬件的同步	276
6.2.2 延迟和带宽模型	217	7.5.3 基于软件的同步	276
6.3 交换策略	221	7.6 放松的存储一致性模型	279
6.4 拓扑结构	223	7.6.1 不依赖于同步的放松模型	280
6.4.1 间接网络	223	7.6.2 依赖同步的放松模型	285
6.4.2 直接网络	226	7.7 推测执行中的存储序违反	289
6.5 路由技术	229	7.7.1 乱序执行处理器中的保守存储模型	289
6.5.1 路由算法	229	7.7.2 推测执行中的存储序违反	290
6.5.2 死锁避免和确定性路由	231	习题	292
6.5.3 放松路由限制: 虚通道和转弯模型	232	第 8 章 片上多处理器	299
6.5.4 进一步放松的路由算法: 自适应路由	233	8.1 概述	299
6.6 交换架构	234	8.2 CMP 的基本原理	300
习题	236	8.2.1 技术趋势	300
第 7 章 一致性、同步与存储一致性	238	8.2.2 机遇	301
7.1 概述	238	8.3 核内多线程	302
7.2 背景	239	8.3.1 软件支持的多线程	302
		8.3.2 硬件支持的多线程	303
		8.3.3 块式(粗粒度)多线程	304

8.3.4 交错 (细粒度)		9.2.1 用户级模拟器和全系统	
多线程	308	模拟器	344
8.3.5 乱序执行处理器上的同时		9.2.2 功能模拟器和时钟精确	
多线程	311	模拟器	345
8.4 片上多处理器架构	314	9.2.3 trace 驱动模拟器、执行	
8.4.1 同构 CMP 架构	315	驱动模拟器和直接执行	
8.4.2 基于异构处理器核的		模拟器	347
CMP 系统	320	9.3 模拟器的集成	350
8.4.3 连体处理器核	322	9.3.1 功能优先模拟器的集成 ...	350
8.5 编程模型	323	9.3.2 时序优先模拟器的集成 ...	351
8.5.1 独立进程	324	9.4 多处理器模拟器	352
8.5.2 显式线程并行	324	9.4.1 串行多处理器模拟器	352
8.5.3 事务内存	326	9.4.2 并行多处理器模拟器	353
8.5.4 线程级推测执行	333	9.5 功耗和热量模拟	357
8.5.5 帮助线程	337	9.6 工作负载采样	359
8.5.6 通过冗余执行提高		9.6.1 基于采样的微架构模拟 ...	360
可靠性	338	9.6.2 SimPoint	361
习题	340	9.7 工作负载特征刻画	361
第9章 量化评估	343	9.7.1 理解性能瓶颈	362
9.1 概述	343	9.7.2 合成基准测试程序	362
9.2 模拟器分类	344	9.7.3 预测工作负载行为	362
		习题	363

总 述

在过去 20 年里,我们经历了由半导体集成和互联网的爆炸式增长所驱动的信息革命。早在 20 世纪 60 年代,摩尔定律就预测了半导体器件的性能呈指数级增长。摩尔定律包含几个构想,其中一个是对微处理器的计算能力,摩尔定律预测在固定成本下微处理器的计算能力每 18 到 24 个月翻一番,所以其成本效率(性能和成本的比值)也将呈指数级增长。我们已经看到整个系统的计算能力一直在以同样的速度持续增长,摩尔定律已经经历了时间的考验,时至今日仍然有效。无论是现在还是将来,摩尔定律都将被反复检验,如今很多人认为摩尔定律将很快走向终结,其主要证据就是 CMOS 工艺尺寸的减小已经快要达到极限,即所谓的 CMOS 极限。

除了半导体工艺的进步,多年来,不断改进的芯片设计方法也使微处理器的性能显著提高。从历史角度看,每一代新工艺产品中,晶体管翻转速度和片上晶体管数量都急剧上升,更快的晶体管翻转速度带来了更高的主频,芯片设计也通过改善电路的设计或增加指令执行中的流水级而带来了更高的主频。流水越深,每个流水线阶段执行功能所需的门延迟就越短。更重要的是,近年来大幅度增加的片上资源给芯片架构师带来了新的机会,可以部署各种技术以提高吞吐量,例如在各级硬件/软件栈上挖掘并行性。如何将上述先进工艺所带来的不断增长的资源充分利用好是计算机体系结构领域面临的主要挑战。

计算机体系结构是一门相对“年轻”的工程科学。在 20 世纪 70 年代初期,伴随着并行处理国际会议(International Conference on Parallel Processing, ICPP)与计算机体系结构国际研讨会(International Symposium on Computer Architecture, ISCA)的成功创立,计算机体系结构的学术和研究领域开始发展,显然那时候的并行处理已经成为计算机体系结构的一大焦点。实际上,在 20 世纪 80 年代和 90 年代初,并行处理和并行计算机体系结构是该领域研究人员之间非常热门的话题,学术研究人员使用大量速度慢、价格低的处理单元来构造可扩展的并行系统。和现在的观点类似,当时人们也认为基于单一中央处理单元的系统必然会很快消亡。然而工业界却选择了另外一种方式,20 世纪 90 年代中期,由于微处理器的繁荣,并行系统黯然失色。随后几年,单一 CPU 的微处理器处理速度得到飞速发展,随着摩尔定律的延续,设计师们迅速提高了芯片中的晶体管密度和时钟频率。过去这些增加的晶体管主要用于设计能够在任意给定周期内处理数百个指令的复杂乱序处理器。相比于解决并行系统的编程复杂性问题,工业界宁可选择通过不断提高复杂乱序处理器的时钟频率来满足计算机用户日益增长的计算需求,因为这在当时是一条便捷可行的途径。在商业领域,多处理器仅仅被视为单处理器系统的扩展,并提供一系列拥有多种性价比的机器。

这种情况在 21 世纪早期迅速改变。技术发展趋势转向支持多个 CPU 或多核处理器。在工业界和学术界中,诸如功耗、复杂性、处理器和内存之间日益增长的性能差距等问题再次引发了人们对并行处理和并行体系结构的研究兴趣。如今,计算机体系结构领域已经达成共识:未来所有的微架构都将采取某种形式的并行执行。片上多处理器(Chip MultiProcessor, CMP)就是这样一种新兴的微架构形式,这也是本书的主要关注点之一。

如何设计一个微处理器、微处理器的一部分或者整个计算机系统是计算机架构师的工作。虽然摩尔定律适用于任何设备或系统,并且本书中所涉及的许多技术适用于其他类型的微芯片

(如 ASIC 等), 但是本书特别着重于能够高效执行指令集的指令处理系统和微处理器。

1.1 什么是计算机体系结构

计算机体系结构是一门工程或应用科学学科, 它主要研究如何在给定的工艺限制和软件要求下设计更好的计算机。过去计算机体系结构是指令集设计的代名词, 然而随着时间的推移, 该术语已经涵盖了计算机的硬件组成、微处理器乃至硬件组件级别的整个系统的设计。本书中, 我们采用了默认的“计算机体系结构”的现代定义, 即“计算机硬件组成和计算机设计”, 在提到指令集时, 我们将明确地使用术语“指令集架构”(简称 ISA)。从指令集的设计发展历史可以看出, 指令集是非常稳定的, 现在只有少数仍为工业界所支持, 虽然目前的指令集可能会不断扩展, 但从头开始设计新的指令集是几乎不可能的, 因为开发一个全新的指令集并进行实现的成本无疑非常巨大。本书中, 我们将粗略地介绍指令集架构, 因为这不是我们的主要目标, 我们的主要目标是在给定成本和工艺限制下快速、正确地实现指令集架构的并行计算机体系结构。

计算机系统的设计非常复杂, 需要许多不同工程和科学领域的能力。管理这种复杂性的唯一方法是将设计分层, 以便不同领域的工程师和科学家可以专注于自己能力所及的那一层。图 1-1 给出了现代计算机系统的分层视图, 每一层都依赖于它的底层。特定领域的专家使用高级语言 FORTRAN、C++ 或 Java 等编写应用程序, 程序在用户级调用函数库来实现常见和复杂的用户级功能, 并调用操作系统来实现系统功能, 如输入/输出、存储管理等。编译器将程序源代码编译成机器和操作系统(通过系统调用)可以理解的代码(如目标代码或机器代码), 编译器的设计者只需要专注于解析高级语言语句、优化程序代码, 并将其翻译成汇编或机器代码, 而目标代码则与函数库相连接以实现一系列常见的软件功能。操作系统通过管理拥有复杂功能的软件以及协调多个用户之间高效、安全、透明地共享机器资源来扩展硬件的功能, 这就是内核开发人员的工作。指令集架构 (ISA) 则位于复杂软件层的下层。



图 1-1 计算机系统的分层视图

ISA 是一个特别重要的接口, 它将软件与硬件、计算机科学家与计算机/电子工程师区分开来。ISA 的实现独立于它之上运行的所有软件层, 计算机架构师的目标是在给定的工艺限制下设计一个尽可能高效地实现指令集的硬件设备, 由于计算机架构师负责设计软硬件的交互层, 因此必须对软硬件都十分熟悉, 与此同时, 他必须了解编译器和操作系统, 还需关注工艺限制和电路设计技术。

系统功能既可以通过硬件实现, 也可以通过软件实现。对于某些类型的异常, 如虚存系统中的 TLB 出现缺失时, 就可以通过硬件或内核软件来完成相应的处理; 而 cache 一致性的某些组件也可以通过硬件或软件来实现。使用软件实现系统功能可以简化硬件电路的设计, 但软件实现的系统功能通常比硬件实现的功能要慢。一旦硬件架构确定了, 就可以交给电路工程师进

行实现，当然这个过程可能需要多个迭代。最后，硬件板卡的设计和开发是由工艺制造工程师和材料科学家共同完成的。

通过分离软硬件层次，ISA 在计算机行业所取得的巨大成就中扮演了至关重要的角色。在 20 世纪 50 年代到 60 年代初，每一代新计算机都有不同的指令集设计，事实上指令集是定义每代计算机设计的标志，但这种策略的缺点是软件很难从一台机器移植到另一台机器上。那时候还没有编译器，所有的程序都是由汇编语言编写的。1964 年，IBM 公司推出 System/360 ISA 而转型成为我们所熟知的大型计算机公司，从那时起，IBM 公司保证其所有未来的计算机都能够运行 System/360 ISA 编写的软件，因为他们会永远支持 System/360 的指令，这就是向后兼容。向后兼容确保了所有二进制编写的代码或是基于 IBM System/360 ISA 的编译可以永远运行在任何 IBM/360 的系统上，因此软件不会过时。IBM 360 指令集在未来可能会有所扩展，且事实的确如此，但它永远不会删减指令，也不会改变任何现有指令的语义或影响。这一战略经受住了时间的考验，即使现在大部分程序都是用高级语言编写然后再编译成二进制文件的，这一点依然很重要，因为二进制对应的源代码可能会丢失，而且通常软件供应商也只提供目标代码的形式。多年来，由于指令的扩展，System/360 先后更名为 System/370、System/390，现在叫作 System z。

由于指令集随时间的改变并不大，因此计算机架构师的主要职责是构建最好的硬件体系结构以满足日益增长的软件系统需求。图 1-2 显示了日益增长的软件需求和硬件性能之间的协同作用，伴随着应用的升级，用户对硬件的要求越来越高（如处理速度、存储容量或输入输出带宽），另一方面，硬件的发展也给软件开发人员提供了更多利用硬件资源的新机会。多年来，正是这种协同作用帮助 Intel 和 Microsoft 公司创造了一个个奇迹。

当前，我们正处在这样一个持续循环中的重要时刻，微架构的演变决定了软件必须进一步并行化，这样才能充分利用多核处理器所带来的新硬件机遇。今天的技术决定了我们必须借助于片上多处理器（CMP）这样的方式才能获得更高的性能，相比于所有的技术挑战，高效并行软件的开发可能是未来计算机系统所面临的巨大挑战。软件必须利用好处理器架构，过去人们已经多次尝试了并行编程以及如何将串行代码编译成并行代码，只有软件可以充分利用和发挥多核、多线程的力量，信息革命才会真正完成。

由于工艺限制，硬件无法如摩尔定律所预见的那样维持单线程的性能呈指数级增长，未来的微处理器将有多个处理器核并行地运行多个线程。未来，单线程性能的平均增长速度将更加平缓，摩尔定律的应用计算能力将会通过在每个处理器节点上并行地运行越来越多的线程来满足。

1.2 并行体系结构的基本组成

图 1-3 给出了一个最基本的个人计算机（PC），它也是一个最简单的并行计算



图 1-2 应用增长与硬件性能之间的协同作用

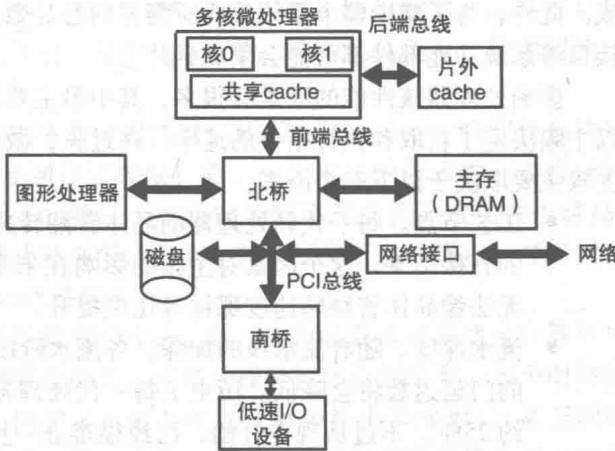


图 1-3 个人电脑的基本架构

机。北桥芯片是系统总线，用来连接处理器、内存和 I/O 设备。PCI (Peripheral Component Interconnect) 是输入/输出总线，用来将高速 I/O 接口连接到磁盘和网络，将低速 I/O 设备 (如键盘、打印机和鼠标) 连接到北桥。南桥是用来连接低带宽外围设备 (如打印机、键盘) 的总线。

图 1-4 给出了一个通用的高端并行体系结构。若干处理器节点通过互连网络连接并相互传送数据。每个节点都包含一个 (可能为多核) 处理器 (P)、一个共享内存 (M) 以及一个缓存层次结构 (C)，处理器节点都连接到全局互连总线或通过网络接口 (NI) 进行点对点互连。计算机系统的另一个重要组成部分是 I/O，I/O 设备 (如磁盘) 通常连接到 I/O 总线，这也是每个处理器节点内部存储之间相互连接的接口。处理器、存储层次结构以及互连网络是并行系统的三个关键组件。

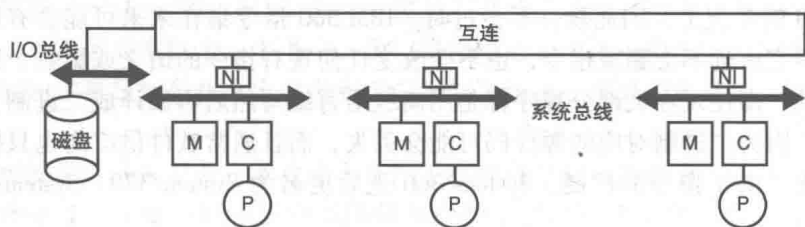


图 1-4 分布式存储的通用多处理器系统

1.2.1 处理器

在这个片上多核处理器和多线程处理器核的时代，首先给出一些基本定义：

程序 (也称为代码、代码段或代码片段) 是程序员所编写的用来执行算法计算步骤的一组静态语句。进程或线程是嵌入执行这些计算步骤的抽象概念。从某种意义上说，若用烹饪来作比喻，程序就是用来说明烹饪过程的食谱。有时进程与线程会交替使用，但通常管理线程比管理进程更容易 (即开销更少)，本书中，我们主要使用线程这个词。

线程在处理器核或中央处理器 (Central Processing Unit, CPU) 上运行，处理器核或 CPU 是能够执行线程指令的硬件实体。一些处理器核是多线程的，可以同时执行多个线程，在这种情况下，每个运行在处理器核上的线程都有其上下文环境。微处理器或处理器由一个或多个处理器核组成，多核微处理器有时也称为片上多处理器或 CMP (Chip MultiProcessor)，多处理器是一组连接在一起执行一个共同工作负载的处理器。

现在的处理器都是批量生产的，微处理器成品由一个或多个处理器核以及多级 cache 组成。此外，为了减少整个系统集成所需要的芯片数量，内存控制器、外部 cache 目录以及网络接口等系统功能部件都可能会集成到片上。

影响处理器核性能的因素有很多，其中最主要的是主频。因为处理器核是流水工作的，所以主频决定了获取和执行指令的速率。在过去，微处理器的性能主要是通过其主频来表示，而主频主要取决于以下三个因素：

- 工艺节点。每一代新处理器的晶体管翻转速度会增加 41%，这是晶体管工艺尺寸减小的直接结果。这个因素对主频的影响在未来将会减弱，因为信号在电线上的传输速率无法像晶体管翻转速度那样等比例提升。
- 流水深度。随着流水级的加深，各流水阶段实现的功能就没那么复杂了，因此各阶段的门延迟数将会降低。历史上每一代处理器各流水阶段的门延迟数都较上一代降低了约 25%。不过从现在开始，已经很难进一步增加流水深度了，因为在不到 10 个门延迟的时间内很难实现真正的流水级功能。

- 电路设计。通过设计更好的电路结构可以进一步改善门延迟和互连延迟。

图 1-5 显示了自 1990 年以来, Intel 各代处理器的最高时钟频率。我们对时钟频率曲线和两个指数函数曲线进行了对比, 一个每年增长 19% (每 48 个月翻一番), 另一个每年增长 49% (每 21 个月翻一番)。19% 的曲线显示了仅由工艺进步所产生的时钟频率增长 (两年更新一代, 每代增长 41%), 它表示保持硬件结构不变, 仅通过工艺升级所能获得的时钟频率增长速度。从 1990 年到 2002 年, 时钟频率的增长速度加快, 不到两年就翻一番 (接近 49% 的曲线), 2002 年之后, 时钟频率增长速度又开始减慢, 并在 2005 年时频率达到顶峰。2003 年以前, 10GHz 的时钟频率看起来很快就会实现, 当时一些人预测在 2010 年之前时钟频率将达到 10GHz。事实上, 如果时钟频率一直以约 49% 的速度增长, 那么在 2008 年时就将达到 30GHz! 但在 2004 年 11 月, Intel 宣布取消时钟频率 4GHz 的奔腾 4 处理器的计划, 转而研究多核微架构。这一声明被认为是微处理器工业界脱离单处理器流水设计转而研究多核微架构的一个重大转折点。

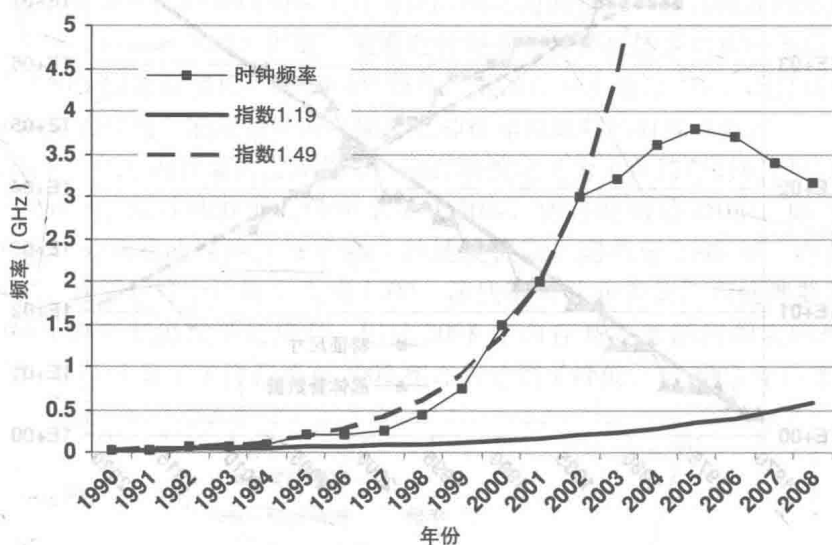


图 1-5 1990 ~ 2008 年 Intel 各代处理器的最高时钟频率变化曲线

据观察, 在 1990 ~ 2002 年, 体系结构在时钟频率的大规模增长中起到了至关重要的作用。这些时钟频率增长在很大程度上归功于奔腾 III 和奔腾 4 微架构中出现的更深流水级设计。为了支持 10 ~ 20 级的流水, 必须从指令流中提取大量的并行成分, 本书涵盖了体系结构的诸多创新, 如分支预测、寄存器重命名、重排序缓冲、无锁 cache、内存相关性预测等, 这些都是高效乱序执行、推测执行以及挖掘大量指令级并行 (ILP) 的关键。没有这些创新, 处理器流水级做得再深也将是徒劳的。

有观点认为过去时钟频率的增长速度在未来将难以持续, 原因有三: 第一, 我们很难构造出不到 10 个逻辑门的有效流水级, 现在已经到达了极限; 第二, 在未来的技术中, 连线延迟而不是晶体管翻转速度将决定时钟周期的大小; 第三, 电路时钟频率越高将导致功耗也越高, 而现如今我们已经达到了单芯片微处理器的功耗极限。图 1-5 有效证实了这一观点, 可以看出自 2002 年以来, 微处理器时钟频率的改进始终停滞不前。

计算机体系结构的贡献不仅仅体现在时钟频率的改进, 指令的吞吐量也可以通过计算机体系结构的改进来得到提高, 如: 更好的存储系统设计, 提高处理器各部分的效率, 每个时钟周期提取和译码多个指令, 在一个处理器核上同时运行多个线程 (称为核内多线程), 甚至同时在多个核上运行多个线程等。除了更高的频率, 每一代新工艺都为计算机架构师提供了可以用

来进一步提高机器性能的大量新资源（如晶体管和引脚），利用日益增长资源的最简洁明了的方法就是在芯片上增加更多的缓存空间，不过这个资源也可以用于其他方面，这也为计算机架构师提供了新的机会。

图 1-6 给出了 Intel 从 1971 年到 2008 年并预测至 2020 年芯片特征尺寸的进化过程。在过去的 20 年里，每两年推出一代新工艺，特征尺寸以每年 15% 的速度缩减，即每一代缩减 30%，每 5 年缩减一半。图 1-6 还显示了自 1971 年以来，每年 Intel 微处理器芯片上的最大晶体管数量，这个数量影响了晶体管密度和晶片面积的增长，从图中可以看到片上晶体管的数量每两年翻一番。截至 2008 年，片上晶体管数量已达到了 10 亿，如果按照这一趋势发展下去，到 2020 年片上晶体管数量将达到 1000 亿。不过，这种趋势是根据过去的的数据所建立的，因此它也只能代表过去的发展趋势。

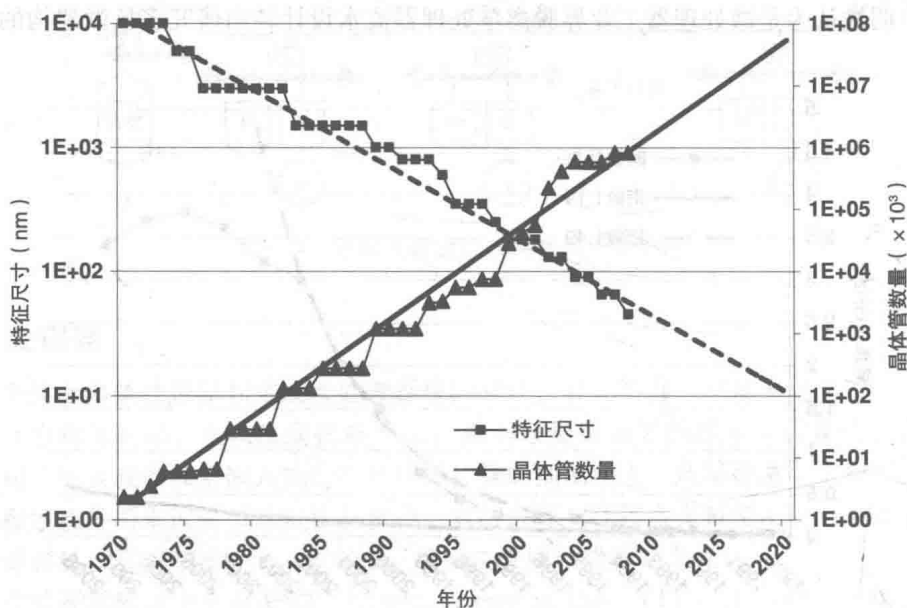


图 1-6 Intel 微处理器的特征尺寸

在今后的十年中，设法以最好的方式开发利用 1000 亿个片上晶体管是计算机体系结构研究领域面临的重大挑战之一，最可能且最有前途的研究方向是在大规模芯片上实现拥有成百上千处理器核的多处理器。

1.2.2 存储

存储系统由高速缓存、主存和磁盘存储组成。任何由处理器直接访问的数据或指令都必须存储在主存中。主存（访问时间在 100ns 范围内）与处理器（时钟频率为几 GHz）之间以及磁盘（访问时间以 ms 为单位）与处理器之间的速度差距是计算机系统领域中长期存在的问题。

存储系统的设计是由它的成本和物理约束所决定的。物理约束包括两方面，一方面是计算机系统需要一个非常大的非易失性存储器来存储永久文件，最高效的半导体存储器（如主存储器和高速缓存）是易失性的，切断电源时它们所存储的内容将会丢失。通常使用硬盘来实现非易失性存储，固态硬盘（SSD）虽然成本较高，但也经常在一些系统中使用。另一方面，访问时间会随着存储空间的增长而增长，未来的技术更是如此，因为访问半导体存储的时间是由线路延时来决定的，存储空间越大，地址译码、地址线传播和位线传播所需的时间就越长。表 1-1 列出了一台 2008 年的个人电脑中不同存储层次的基本成本和规模大小。

表 1-1 2008 年一台个人电脑存储的基本规模与成本

存储	规模	边际成本	每 MB 成本	访问时间
二级 cache (片上)	1MB	\$20/MB	\$20	5ns
主存	1GB	\$50/GB	5c	200ns
磁盘	500GB	\$100/500GB	0.02c	5ms

存储分层的目的是让处理器认为整个存储系统的平均访存时间近似于处理器时钟周期时间，同时使得整个存储系统的存储空间和单位存储的成本接近于磁盘存储器。

主存

主存（由 DRAM 组成）与处理器之间的速度差距大到足够影响处理器的性能。例如，若处理器的时钟频率是 1GHz，主存的访问速度是 100ns，则处理器在等待访存期间可以执行 100 条指令，无论处理器多复杂、主频多快，它的速度都无法超过存储系统提供指令和数据的速度。

从历史上看，处理器的处理周期和主存访存时间之间的差距一直以惊人的速度在增长，这一趋势称为内存墙（memory wall）问题。随着时钟频率和计算机体系结构的创新，微处理器的速度每年以超过 50% 的速度增长，而 DRAM 每年性能的提高不超过 7%，而且访问时间不仅包括访问 DRAM 芯片的时间，还包括经由存储总线和存储控制器的时间延迟。

图 1-7 给出了多年来内存墙的发展趋势。内存墙被定义为主存访问时间同处理器时钟周期时间的比值。1990 年，Intel i486 的时钟频率是 25MHz，访问时间是 150ns，则内存墙值为 4。如果从 1990 年起，处理器的性能以每年 49% 的速度提高，则截至 2008 年，内存墙将增长至 1990 年的 400 倍，也就是这个比值会变成 1600。显然事实并非如此。当处理器的时钟频率达到顶峰时，DRAM 的速度仍在平稳提升，因此 2008 年内存与处理器间的实际性能差距仅是 1990 年的 40 倍。图 1-7 显示了内存墙在 2002 年左右达到了峰值，自 2003 年以来则有所下降。

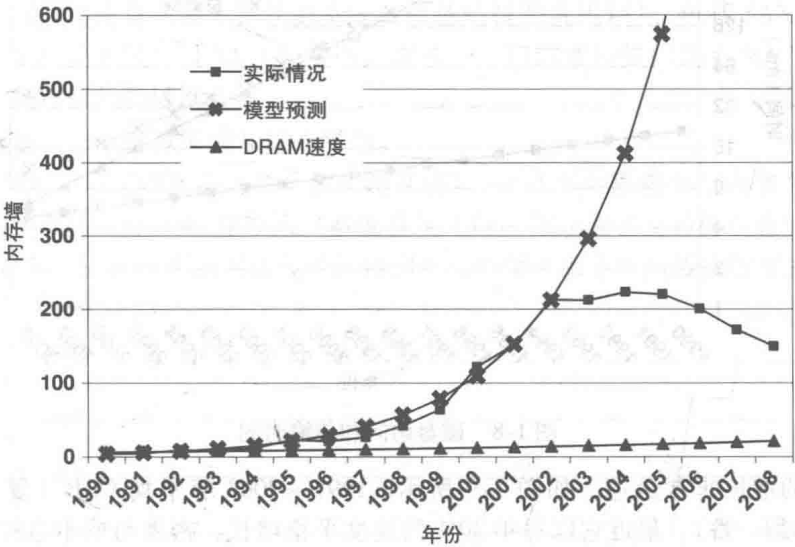


图 1-7 内存墙（DRAM 访问时间/CPU 时钟周期时间）

从历史上看，通过在处理器和主存之间构建层次化的 cache 结构，以及在处理器和层次化的 cache 结构中构建较大 cache miss 时的延迟容忍机制，可以掩盖 DRAM 的平庸性能。层次化的 cache 结构是由几个不同大小和不同访存时间的缓存层次所构成的，层次化的 cache 结构依赖于访存的位置属性。

显然，主存和处理器之间的速度差距至今仍然是一个严重的问题，但如果照目前的趋势发

展下去，内存墙在未来不会进一步加大，至少它的增长速度将大大降低。目前达成的共识是内存墙问题已经控制得很好，随着片上多处理器的出现，以及对处理器节点 cache miss 所带来的延迟的积极优化，目前，相较于存储访问延迟，存储带宽（存储系统单位时间内的访问次数）正快速成为我们所面临的更主要的问题。

虽然 DRAM 的速度并未随着时间的推移获得大幅提高，但每个 DRAM 芯片的存储位数却以每 3 年 4 倍的速度增长。1977 年，单个 DRAM 芯片的存储容量是 1Kbit，1992 年和 2007 年单个 DRAM 芯片的存储容量分别达到了 1Mbit 和 1Gbit，如果照这个趋势发展下去，到 2021 年，单个 DRAM 芯片的存储容量将达到 1Tbit (10^{12} bit)。

磁盘

磁盘访问时间由两部分组成：一部分是访问时间（寻道时间加延迟时间），它独立于传输数据的规模；另一部分是传输时间，它与传输数据的规模成正比。寻道时间是指磁头移动到指定磁道上所经历的时间，延迟时间是指磁头到达磁道上第一个要访问的记录时所经历的时间，延迟时间和传输时间都取决于磁盘的旋转速度。图 1-8 显示了访问一块 4KByte 数据所需的平均访问时间与传输时间。过去磁盘访问时间主要由传输时间所主导，然而多年来当访问时间以每年 8.5% 的速度缓慢减少时，传输时间却以每年 40% 的速度大幅度减少，如果传输时间仍以当前速度减少，则磁盘访问时间将主要由访问时间所主导。需要注意的是这里提到的所有时间仍以 ms 为单位，图中给出了访问时间随着年份的减少速度，可以预见未来处理器与磁盘性能之间的差距仍将非常大。

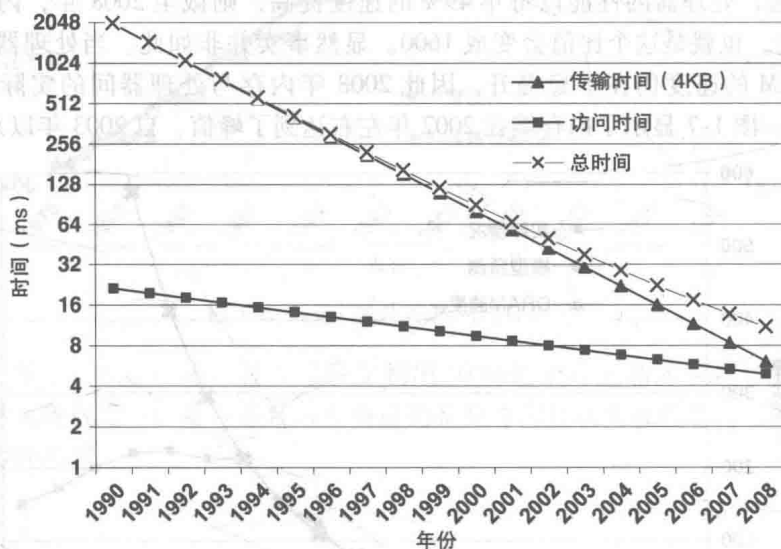


图 1-8 磁盘访问和传输时间

磁盘密度的增长非常迅速，如图 1-9 所示，1997 ~ 2003 年平均 CGR（复合增长率）是 100%（即每年翻一番），最近它以每年 30% 的速度平稳增长，密度与成本直接相关，因此单位存储的成本也在迅速下降。

由于磁盘访问速度为机械设备的工作速度，因此辅助存储器和半导体存储器之间的速度差距也非常大。如果一次磁盘访问需要 10ms，处理器的时钟频率超过 1GHz，则在磁盘访问期间可以执行数以百万计的指令，因此处理器从不等待磁盘访问的完成，通常是多个进程之间共享包括主存在在内的硬件资源。当正在运行的进程请求访问磁盘时，操作系统会先阻塞它，并调度执行另一个处于就绪态的进程，一旦磁盘访问结束，之前阻塞的进程将切换为就绪状态，以便再次被操作系统调度执行。

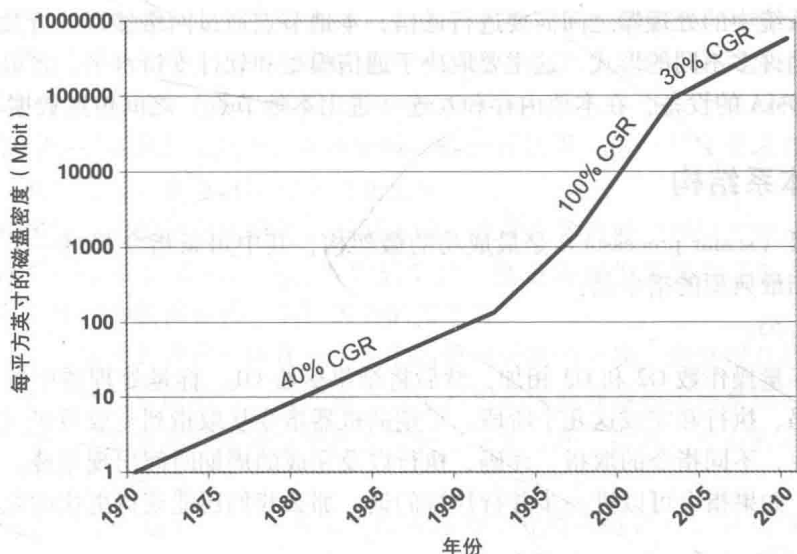


图 1-9 磁盘密度

1.2.3 互连

互连网络用来连接各级组件。

- 片上互连用来传送不同流水级和执行单元之间的数值，现在片上互连也用来连接各个处理器核到共享的 cache bank。
- 系统互连用来连接各个处理器（CMP）和存储器以及 I/O 设备。
- I/O 互连用来将各种 I/O 设备连接到系统总线上。
- 系统间互连用来连接各个独立的系统（单独的机架或机箱），包括 SAN（系统区域网，连接短距离的系统）、LAN（局域网，连接一个组织或建筑内的系统）和 WAN（广域网，连接远程的局域网）。
- 互联网是一个全球范围内互连的网络。

本书中我们关注系统内互连（片上和系统互连），并且不会直接介绍由多个系统互连的任何网络。图 1-10 表明，自 2000 年以来，系统总线（Intel 微处理器上的前端总线）速度已经大幅提升了一个数量级，而这个前端总线的时钟频率与数据进出芯片的传输带宽直接相关。

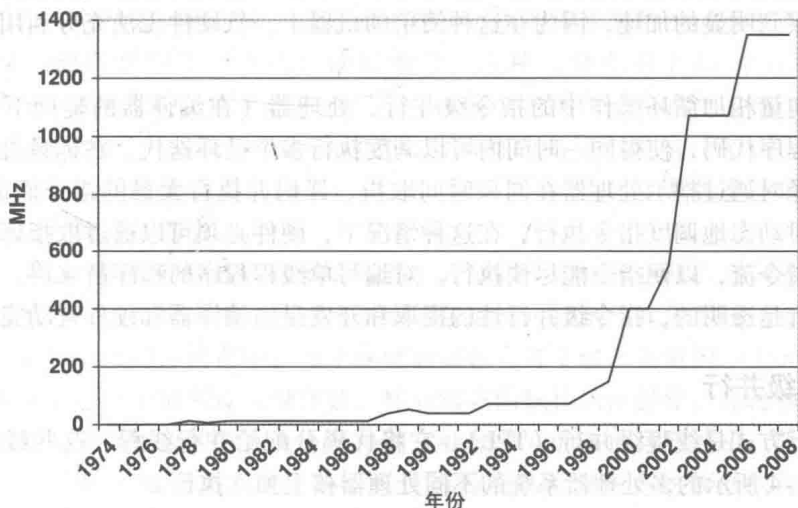


图 1-10 Intel 处理器前端总线速度

多处理器系统中的处理器之间需要进行通信，本地节点通过网络接口进行数据的收发。网络接口可以采用许多不同的形式，这主要取决于通信模型和软件支持水平，它最原始的形式是一个内存映射 DMA 的设备，在本地内存和互连（进出本地节点）之间传递数据。

1.3 并行体系结构

标量处理器（scalar processor）是最成功的微架构，其中每条指令都对一个标量数据元素进行操作。例如最典型的指令是：

```
ADD O1, O2, O3,
```

该指令将标量操作数 O2 和 O3 相加，然后将结果赋给 O1。标量处理器中的每条指令都经历了取指、译码、执行和完成这几个阶段。早期的机器指令从取指到完成只经过一次处理，随着流水线的产生，不同指令的取指、译码、执行以及完成的周期时间出现重叠，从而提高了处理速度。不过，如果指令可以进一步并行执行的话，那么我们还能获得更快的处理速度。

1.3.1 指令级并行

为了使指令可以并行执行，程序必须具备指令级并行（ILP）性，例如指令之间具有一定程度上的独立性。如果指令之间相互依赖，则它们必须串行执行，每条指令必须等待之前指令的执行结果，编译器必须能够挖掘出指令级并行（ILP），并且微架构也必须能够开发和利用这一特征。

图 1-11a 给出了两个小程序片段，左边的是计算两个向量相加，右边的是计算向量各个位的和。向量相加程序的潜在执行速度非常快，因为所有循环迭代是相互独立的，所以指令都可以同时跨循环迭代执行，从而获得 1024 倍的提速。而右边的循环并行是受限的，因为每次循环迭代都依赖于前一次循环变量 S 的累加执行结果，因此直到当前变量 S 更新完成，才能执行下一次更新。相比

```
for (i=0,i++,1023)    for (i=0,i++,1023)
  A[i]:=B[i]+C[i];    S:=S+A[i];

a) 单线程程序

for (i=0,i++,255)    for (i=0,i++,255)
  A[i]:=B[i]+C[i];    my_S:=my_S+A[i];
                      S:=my_S+S;

b) 多线程程序（线程数为4）
```

图 1-11 ILP 与 TLP

比于向量各位数字求和的循环操作指令级并行严重受限，向量相加的循环操作却可以进行大规模的指令级并行。然而，如果两个循环都在一个流水级较少或根本无流水的简单机器上运行，我们将不会观察到明显的加速，因为在这种简单的机器上，软硬件无法充分利用程序中的指令级并行性。

为了利用向量相加循环操作中的指令级并行，处理器（在编译器的辅助下）必须能够同时探测大量的程序代码，使得同一时间内可以调度执行多个循环迭代。在标量处理器中，指令执行可以在编译时通过指示处理器在同一时间取指、译码并执行大量的指令而进行静态调度，也可以在执行时动态地调度指令执行，在这种情况下，硬件必须可以通过取指译码大量的指令来预测未来的指令流，以便指令能尽快执行。对编写单线程程序的程序员来讲，开发静态或动态的指令级并行是透明的，指令级并行性的提取和开发是由编译器和硬件自动完成的。

1.3.2 线程级并行

另一种并行方式是线程级并行（TLP），它将代码分配给并行线程，这些线程可以在多核处理器或如图 1-4 所示的多处理器系统的不同处理器核上独立执行。

图 1-11b 显示了一个四线程程序中第一个线程（即 thread_0）的简化代码片段，它将向量

加代码直接分配给 4 个线程，每个线程执行 1/4 的向量加，因此速度提升 4 倍。向量各位数求和的程序也可以分配给 4 个线程，只是编程上需要做更多的努力，它的主要思想是让每个线程都计算长度为 256 的向量的部分和，并累加到私有变量 my_S 上，当每个线程都执行完，各个线程的结果相加并赋值给全局变量 S，这就是最终的计算结果。这个程序要求各个线程之间存在共享内存，以便所有线程都可以访问全局变量 S。

多处理器和多核系统的指令或时钟是不同步的，线程在各自处理器核上独立运行，因此各线程有时必须进行同步以协调活动或进行数据交换，这通常称为多指令多数据（MIMD）系统，以此强调每个处理器核都在独立执行各自的指令流。

提取线程级并行通常是程序员的任务，他们需要设计新的算法，调整现有代码以体现线程级并行，并负责安排好线程间的通信与同步。为了利用多核处理器和多核系统，运行的负载程序必须进行并行化，这通常意味着需要重新改写整个应用程序。因此，并行程序的生产效率是未来计算机系统的重大挑战之一。

解决并行编程问题的途径之一是采用并行编译器。并行编译器可以自动将单线程程序转化成多线程程序，然而并行编译器还存在以下三个主要缺陷：第一，并行编译器生成的并行代码效率很差；第二，由于指针变量的值在编译时是未知的，因此编译很容易失败；第三，存在一部分算法，对这部分代码进行并行可能不是最好的解决方案，然而并行编译器却对其进行了并行化。因此，通常只有按照多处理器的规模裁剪算法和代码才能获得最佳的性能。

1.3.3 向量和阵列处理器

向量处理器和阵列处理器比标量处理器更善于利用指令级并行，也更善于并行在多个操作数上执行相同的操作，与此同时，向量处理器和阵列处理器也有适用于同时处理大量标量操作数的向量指令，例如：

```
VADD VO1,VO2,VO3,
```

VO1 指定标量运算的一个数据流或向量。例如图 1-11a 中的向量相加可以通过一条向量指令执行，此时 VO1 是 A[0:1203]，VO2 是 B[0:1203]，VO3 是 C[0:1203]。当然，这种向量指令的执行可能需要很多个时钟周期，但对如图 1-11a 所示的程序来说，它的执行效率可能是非常高的，从而为利用向量指令开发硬件并行提供了可能。

向量指令可以在向量处理器或阵列处理器上执行，这两种微架构的区别在于向量处理器利用流水线而阵列处理器利用并行来执行向量指令，这种差异在图 1-12 中进行了说明。在图 1-12a 所示的向量处理器中，两个输入的向量操作数可能被存储在内存或寄存器堆中，控制单元（CU）获取向量指令并分配一个计算流水线对所有的向量元素执行 VADD 操作，向量运算开始后，每个时钟周期向量元素都会进行相加，直到 1024 个时钟周期才停止，结果也以相同的速率存储在内存或寄存器中。

在图 1-12b 所示的阵列处理器中有多个处理单元（PE），每个处理单元都由一个执行单元和一个本地存储器组成，并可以用本地存储器中的数据执行控制单元分配的指令。两个输入向量被交叉存储在处理单元的存储器中，每个存储器都包含各个输入向量的一个元素，控制单元广播指令以在本地寄存器中加载输入操作数，然后对它们执行加法操作，并将执行结果存储回本地存储器，最后结果操作数的值被交叉地存储在处理单元的本地存储器中。阵列处理器也称为单指令多数据（SIMD）处理器，这反映了所有的处理单元对各自的数据执行相同的指令操作。向量和 SIMD 机器都能够利用细粒度的指令级并行。

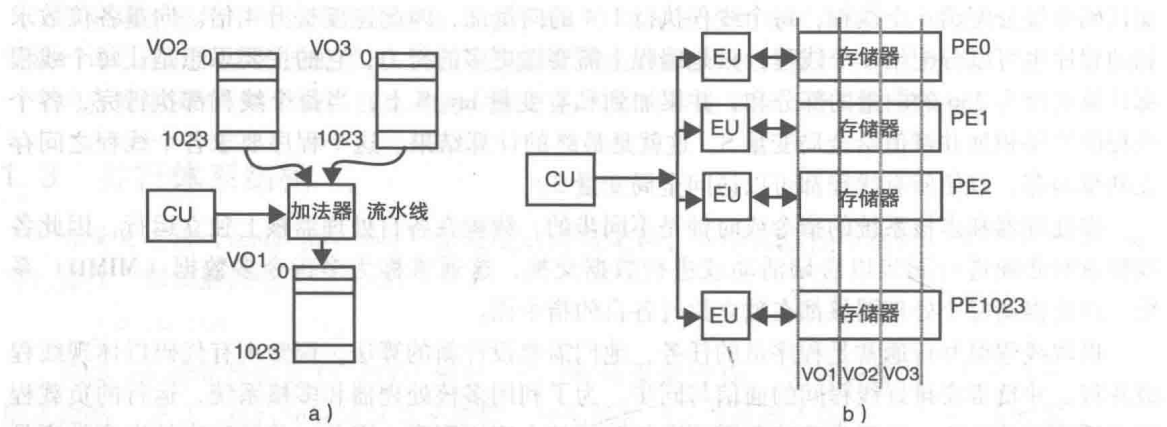


图 1-12 向量处理器 (a) 和阵列处理器 (b) 上的向量指令执行

编译器的支持对提取程序并行性以及将循环转化为向量代码至关重要，然而向量处理器已经存在很长时间，并且它们的编译器技术已经非常成熟，即使是图 1-11a 中的向量元素循环求和也可以在向量处理器和阵列处理器上进行向量化。

1.4 性能

性能是用来衡量微架构和系统的最重要标准，以下是两种性能度量标准。

- 执行时间（也称为延迟或响应时间）是过程由开始到结束所对应的墙上时钟经过的时间。执行时间包含过程执行中操作系统花费的时间。
- 系统吞吐量是单位时间内的进程执行数量。

虽然这两种度量标准通常是一致的（例如，执行时间越短，则吞吐量越高），但不一定都是如此。考虑下面的例子，现提出两种将单处理器计算机系统吞吐量提高一倍的办法，一种办法是将单个处理器的运行速度提高到原来的两倍，另一种办法是在原处理器基础上添加第二个处理器，使两个处理器都与原系统具有相同的运行速度。这两个系统都满足吞吐量需求，然而第一种办法比较好，因为它不仅能增加一倍的吞吐量，还将每个进程的执行时间减少了一半。

过去，执行时间被认为是衡量性能的最终度量标准，然而随着多核处理器时代的到来，进程（或线程）的吞吐量变得更加重要，因此未来这两种度量标准都非常重要。由于某些程序不能简单且高效地进行并行化，因此单线程性能仍然很重要，尤其是在比较核心架构的时候。

在基于同步逻辑设计的处理器核上，程序的执行时间取决于三个影响因素的乘积：

$$T_{\text{exe}} = IC \times CPI \times T_c \tag{1.1}$$

其中 IC (instruction count) 为程序执行的指令数，CPI (clock per instruction) 为机器执行程序时执行每条指令所需的平均时钟周期， T_c 是机器的时钟周期时间，该公式直接表明了同步系统的处理单元是一个时钟周期。

当 CPI 可以直接估计时，这个公式非常有用。然而，由于当前复杂的处理器和层次化的 cache 结构，要针对每条指令来确定 CPI 是几乎不可能的。事实上，CPI 不仅取决于一条指令的执行，还取决于同一时间执行的所有指令，因为这些指令执行是相互干扰的。换句话说，指令的 CPI 取决于它执行的上下文环境。如果我们将上面的公式颠倒过来，它仍然有用：

$$CPI = T_{\text{exe}} / (IC \times T_c) \tag{1.2}$$

执行时间和程序指令数可以通过运行程序或模拟其执行得到。对给定的时钟频率和程序指令数来说，平均 CPI 是用来描述机器上程序执行速度的非常直观的度量标准。由于 CPI 比执行时间更加直观，因此常被用来比较拥有相同指令集和相同工艺的不同体系结构。

当处理器核在同一周期能够获取并开始执行多条指令时, IPC (instruction per clock) 是更方便的度量标准。

$$\text{IPC} = 1/\text{CPI} \quad (1.3)$$

在此时的上下文环境中, IPC 比 CPI 更好, 因为它与性能呈正相关, 而 CPI 与性能呈负相关, 并且 IPC 通常是一个大于 1 的整数, 而 CPI 通常小于 1。

目前处理器设计日趋复杂, 处理器可以改变时钟频率, 具有复杂的 cache 层次和微处理器结构, 有效地应用这些简单的公式将变得更加困难, 但由于其简单, 因此依旧非常有价值。

1.4.1 基准测试集

单线程性能的最终度量标准是执行时间, 因此就必须决定要执行的程序。显然, 比较两台机器的性能需要在两台机器上运行相同的程序, 然后比较执行时间。测试程序应该在源代码级别指定, 因为机器的性能也取决于其编译器的质量, 有些机器在设计时会充分考虑编译器, 使之更容易对代码进行优化。相对于 ISA, IC 和 IPC 更多地取决于编译器, 但存在的问题是一个体系结构 (和它的编译器) 可能是基于一个特定的程序设计并调试的, 对其他程序来说性能可能会很差, 因此, 我们需要采用更加严格的标准来选择能够代表某一应用领域程序行为的多个程序, 这就是基准测试集的作用。基准测试集可覆盖通用应用程序、科学和工程应用程序、商业应用程序 (例如, 数据库、Web 服务器、在线交易处理)、多媒体应用程序, 以及图形和嵌入式应用程序等。

SPEC (Standard Performance Evaluation Corporation) 基准测试集常用来比较个人电脑、工作站以及微处理器的性能。SPEC 程序集由两个子测试集构成: 整型测试集 (例如, 编译器、数据压缩、语言解析器、模拟仿真等) 和浮点测试集 (例如, 偏微分方程求解器、图像等)。它有两个用来总结和表征每类应用机器性能的指标: SPECInt 和 SPECfp。SPEC 测试集会不断进行更新, 这主要是因为机器性能增长得非常快。例如, 一个基准测试程序在设计时, 典型的 cache 大小是 128KB, 它无法测试拥有 4MB cache 的微处理器的内存性能。因此随着时间的推移, 就出现了几代 SPEC 基准测试集的典型代表: SPEC89、SPEC92、SPEC95、SPEC2000 和 SPEC2006 (目前的基准测试集)。基准测试集从一代到下一代发展时, 既会删除或修改一些程序, 也会加入一些新的程序。

此外还有一些重要的用于表征大型服务器性能的基准测试集, 其中包括 TPC (Transactional Processing Council) 基准测试集。现存几类适用于不同类型服务器的 TPC 基准测试集: TPC-B 和 TPC-C 主要用于在线事务处理 (OLTP), TPC-D 和 TPC-H 主要用于决策支持系统 (如商业管理数据库), TPC-W 主要用于 Web 服务器。最后, 嵌入式系统可以使用 EEMBC 或 MiBench 测试集进行评估, 而系统的多媒体功能则可以通过 MediaBench 进行比较。

报告一组程序的性能

现在一个棘手的问题是如何用单个性能数字来反映一整套测试程序的性能, 一种解决方法是取所有执行时间的算术平均值:

$$\bar{T} = \frac{1}{N} \times \sum_{i=1}^N T_i \quad (1.4)$$

在这个公式中, 执行时间最长的程序可能会减弱其他短程序的效果, 而这些短程序可能是整个体系结构中最高效的部分, 加权平均可以反映不同程序的不同重要性。例如, 每个执行时间可以由与执行指令数成反比的影响因子 (ω_i) 或反映每个程序执行频率的影响因子进行加权。

$$\bar{T} = \frac{1}{N} \times \sum_{i=1}^N T_i \times \omega_i \tag{1.5}$$

另一种提高公平性的方法是执行指令数相同的样本程序，Simpoints 就是采用这样的方法。在 Simpoints 中，每个基准测试程序都使用一个或多个典型的执行片段，这些片段包含相同的指令数。Simpoints 的另一个目标是降低基准测试集的模拟时间，因为完整执行这些基准测试程序需要花费很长的时间。Simpoints 使得每个代表基准测试程序执行的片段都具有相同的指令数，因此，较长的基准测试程序也不会过度影响最终的度量值。关于工作负载采样技术的内容（包括 Simpoints）都将在第 9 章中进行讲解。

报告加速比

一般情况下，对于给定程序，被评估机器相对于基准机的加速比定义如下：

$$S_i = \frac{T_{R,i}}{T_i} \tag{1.6}$$

其中， $T_{R,i}$ 是程序 i 在基准机上的执行时间， T_i 是程序 i 在被评估机器上的执行时间。以单个性能数字来报告一组测试标准结果的常见方法是计算加速比的某种形式的平均值。一种方法是计算加速比的算术平均值：

$$\bar{S}_a = \frac{1}{N} \times \sum_{i=1}^N S_i \tag{1.7}$$

另一种方法是计算加速比的调和平均值：

$$\bar{S}_h = \frac{N}{\sum_{i=1}^N \frac{1}{S_i}} \tag{1.8}$$

使用加速比的算术平均值或调和平均值都存在一些问题，主要问题在于，在相比较的两台机器中选择不同的机器作为基准机时，可能会产生不同的结论。而几何平均值不存在这个问题：无论我们选择哪个做基准机，两台机器之间的相对速度始终是相同的，简单地改变基准机的话不会得出机器相对速度不同的结论。加速比相对于基准机的几何平均值由下式给出：

$$\bar{S}_g = \sqrt[N]{\prod_{i=1}^N S_i} \tag{1.9}$$

几何平均值的另一个优点是它的可组合性。如果我们分别获得两组单独程序加速比的几何平均值，则这两组程序总加速比的几何平均值可以由各组平均值相乘得到，即每个程序对平均值的贡献是独立于其他程序的贡献的，并且独立于程序的执行长度。

通常情况下，无论采取什么方法来计算平均值，相对于基准机的平均加速比都会存在很多问题，下面给出一些具体的例子说明。

例 1.1 表 1-2 给出了程序 A 和程序 B 在测试机 1 和测试机 2 以及在基准机 1 和基准机 2 上的执行时间，使用执行时间算术平均值的比例和相对于基准机 1 和基准机 2 的加速比平均值（算术平均值、调和平均值、几何平均值）来比较测试机 1 和测试机 2。

表 1-2 两个程序的执行时间

	程序 A	程序 B	算术平均
机器 1	10s	100s	55s
机器 2	1s	200s	100.5s
基准机 1	100s	10000s	5050s
基准机 2	100s	1000s	550s

表 1-3 的前两列给出了每个测试机相对于每个基准机的加速比，第三列给出了由平均执行

时间的比例计算得到的加速比，最后三列分别给出了程序 A 和程序 B 加速比的算术平均值、调和平均值以及几何平均值。

表 1-3 相对于基准机 1 和基准机 2 的加速比

	程序 A	程序 B	平均加速比	算术平均值	调和平均值	几何平均值
测试机 1 相比于基准机 1	10	100	91.8	55	18.2	31.6
测试机 2 相比于基准机 1	100	50	50.2	75	66.7	70.7
测试机 1 相比于基准机 2	10	10	10	10	10	10
测试机 2 相比于基准机 2	100	5	5.5	52.5	9.5	22.4

首先考虑平均执行时间的比例。测试机 1 以近似 2 倍的比率优于测试机 2，并且这个结论是独立于基准机的。但是，需要注意的是第二台机器执行程序 A 时十分高效，但并没有在最终的算术平均值上体现出来，这是因为执行时间的算术平均值是偏向于最长执行的（程序 B 符合这种情况）。

如果度量标准采用加速比的某种平均值，得到的结论则是不同的，在这种情况下，通常认为测试机 2 是优于测试机 1 的。当度量标准采用加速比的算术平均值时，无论是相对于基准机 1 还是基准机 2，测试机 2 都是优于测试机 1 的，而相对于基准机 2 的比较结果显示测试机 2 以 5 倍的比率优于测试机 1，这显然与基于执行时间的算术平均数得到的结论相矛盾；当度量标准采用加速比的调和平均值时，相对于基准机 1，测试机 2 以近似 4 倍的比率优于测试机 1，相对于基准机 2，测试机 1 只略优于测试机 2；当度量标准采用加速比的几何平均值时，无论是相对于基准机 1 还是基准机 2，测试机 2 都以近似 2.24 的比率优于测试机 1。

当度量标准采用加速比的算术平均值或调和平均值时，得出的结论是依赖于基准机的，而基于加速比的几何平均值得出的结论是独立于基准机的。但需要注意的是，加速比的几何平均值与基于执行时间的总和或加权总和得到的性能并不成正比。

1.4.2 Amdahl 定律

Amdahl 定律在 1967 年由 Gene Amdahl 首次提出，Gene Amdahl 是 IBM 的工程师，他参与了 IBM System/360 的构思与实现。Amdahl 定律最初是针对并行体系结构提出的，但它也适用于任何对计算机系统性能的增强与提高。除了计算机，它也适用于其他度量，如功耗或可靠性，甚至包括许多的人类活动。Amdahl 定律非常简单，但过去凡是忽略了该定律的计算机架构师都为此付出了惨痛代价。

首先，我们来推导 Amdahl 定律的最常见形式。如图 1-13 所示，假设对现有系统进行优化改进，用 E 表示，使得计算占比为 F 的部分速度提高了 S 倍，这里的优化既包含硬件优化（如增加了浮点协处理器）也包含软件优化（如新的编译器优化）。

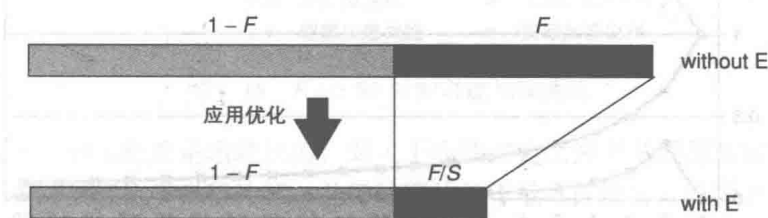


图 1-13 Amdahl 定律

设 T_{exe} 为执行时间，优化后总计算执行时间如下：

$$T_{\text{exe}}(\text{with E}) = T_{\text{exe}}(\text{without E}) \times [(1 - F) + F/S] \tag{1.10}$$

通过优化所获得的加速比如下：

$$\text{speedup}(E) = \frac{T_{\text{exe}}(\text{without } E)}{T_{\text{exe}}(\text{with } E)} = \frac{1}{(1 - F) + F/S} \tag{1.11}$$

式 (1.11) 是 Amdahl 的加速比定律，从这个加速比定律中我们可以得到以下经验启示。

启示 1：可能获得的最大加速比受限于不能被加速的代码部分，最大加速比为 $1/(1 - F)$ ，这是一个残酷的结论。我们看一个例子，假设经过几周紧张的研究，工程师发明了一种计算实数正弦函数的变革性方式，这个发明十分巧妙，它使计算正弦函数的执行时间为原来的 $1/1000$ ，而发明者对他的工作也感到十分自豪，并申请了专利。然而，由于正弦函数在科学工作总执行时间中所占的比例不到 $1/1000$ ，因此他的发明带来的加速比小于 $1/0.999$ 或 0.1% 。这个发明只有对专门用于计算正弦函数的机器来说是有用的，如果这样的机器非常有用，那么发明者才会从这个专利中获利。

启示 2：优化常见的情况。致力于提高计算机系统性能的人应该关注那些在执行时占用最多时间的组件，并尽力缩短其执行时间。试图加速那些只在特定执行中占用较短时间的功能部件是徒劳无功的。例如，在启示 1 正弦函数的例子中，更好的方法是将正弦函数作为一个软件子程序来调用执行，而不是设计专用硬件来对其进行加速。一般来说，存储 (cache) 访问等常见功能是通过硬件来实现的，而罕见功能则通常是由软件来实现的。幸运的是，事实证明最耗时的功能往往都是简单的。通过软件来实现特定功能的一种重要机制是异常，当硬件需要软件协助时，就触发一个异常，这个异常指导处理器执行异常处理程序，异常处理程序扩展了硬件的功能。举例来说，没有必要直接用硬件来处理缺页，而且由硬件来做的话其过程极其琐碎和复杂，而通过异常来触发硬件在内核中执行一个缺页处理程序则要简单和方便得多。

启示 3：收益递减的规律。一旦我们基于常见情况规则选定了某个候选的加速功能，就没有必要无限制地投入资源来提高该功能的加速比。图 1-14 给出了当 $F = 0.5$ ，加速部分的速度提高倍数 S 从 1 增加到 10 时，Amdahl 定律的加速比，可以看到最大的加速比为 2，对每个 S 值的边界加速收益和剩余潜在加速收益也在图中列了出来。 S 的每一次增加都需要投入额外的资源，当 $S = 2$ 时，获得了 33% 的速度提升，且速度再提升 0.66 仍然是可能的，在此之后，对每个 S 值来说，边界加速收益迅速下降，当 $S = 5$ 时，边界加速收益为 6.67%，而剩余加速收益

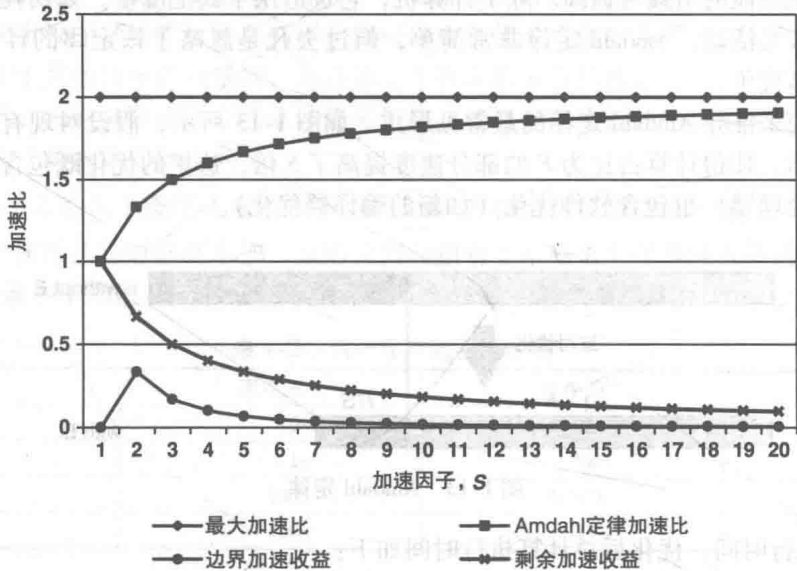


图 1-14 收益递减规律图解

为33%。显然,随着 S 的增加,继续投入资源以进一步提高速度的收益越来越少,而潜在的加速收益迅速降低到极值点,此时继续追求这种设计收益显然意义不大。一旦获得了大部分的加速收益,计算机架构师就应该重新评估常见功能部件,并寻找其他机器功能的加速收益。

并行加速

在一个多处理器系统上增加处理器个数或在片上多处理器系统中增加核数被视为一种增强单核性能和 Amdahl 加速比的有效方法。在这种情况下,加速倍数等于处理器或核的数量。设 P 为处理器或核的数量, F 是在 P 个处理器或核上可并行代码部分所占的比例,可得出如下公式:

$$S_P = \frac{T_1}{T_P} = \frac{1}{1 - F + F/P} = \frac{P}{F + P(1 - F)} < \frac{1}{1 - F} \quad (1.12)$$

可获得的最大加速比受限于不能被并行化的代码部分(如 $1 - F$),这一结论非常残酷。例如:即使我们部署了上百万个处理器或核,使得99%的执行可以被并行化,整体的加速比也不可能超过100。直观地说,如果部署 P 个处理器用于解决问题,假设运行在多处理器和单处理器系统上的工作是相同的,则最大加速比应为 P ,因此 P 被称为理想加速比。

图1-15给出了 $F = 95\%$ 时的不同加速曲线。除了 Amdahl 加速曲线以及加速比恒为20的最大可能加速曲线外,图中还列出了理想加速曲线和一个经历增长、峰值、下降全过程像炮弹射击轨迹的加速曲线,而这种曲线是通过实践观察得到的。 P 的数目持续增加,使得每个线程的计算量减少,则并行带来的速度提升被线程间通信开销所抵消,这时就产生了这种曲线。由此得出:当到达某个临界点时,如果继续添加更多的处理器核进行计算,那么对处理器的性能是有害无益的。

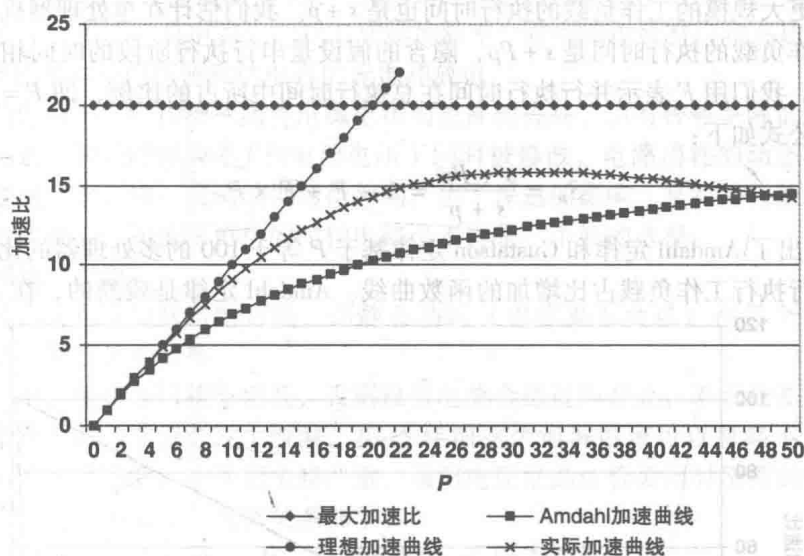


图 1-15 $F = 0.95$ 时的可能加速曲线

现在的问题是,加速能否是超线性的,即大于线性加速比为 P 的理想加速。这似乎违背了守恒定律,然而有时候相比于串行计算,并行计算的工作量可以更少,实际上人们也确实观测到了超线性加速比的现象,这是由于现如今的处理器和核都配备了至少一级的 cache 结构,增加系统处理器或核数的同时也增加了 cache 存储空间的总量。假设一个应用的数据集非常大,以致于单核层次化的 cache 结构容纳不下,但当应用被分配给越来越多的核时,每个处理器访问的数据集就会减少,当每个处理器访问的数据集可以由其 cache 所容纳时,由于 cache 失效

的减少以及更加高效的内存访问，有可能会使加速值产生质的飞跃。

目前为止，我们实际上隐含假定加速比是针对某个给定算法定义的，因此运行在多处理器和单处理器系统的计算量是相同的。现在的问题是，加速比是否应当调整为在应用层进行定义，一个应用可以通过不同的算法来实现，例如整理文件可以通过快速排序或冒泡排序来完成。若加速比被定义在应用层，当采用不同的算法时，通过串行和并行来实现的计算量可能是不同的，并且比较并行和串行机上的执行时间会变得很困难。在单处理器中，最佳算法是计算量最少的算法，因此，即使单处理器和多处理器都选择了最佳算法，所产生的加速比应该还是小于理想加速比。

Gustafson 定律

Amdahl 定律经常遭到批评，因为它假定要解决的问题规模是固定的。然而，工作负载的规模和复杂性往往是随着处理能力的增长而增长的。考虑到每一代新的片上处理器都将拥有更多的核数，我们可以预期软件应用也将获得相应的增长。如果当前用户对系统运行应用程序的速度感到满意的话，那么进一步提高处理能力则可以支持更加复杂的软件，在几乎相同的运行时间里，提供更新的软件应用，从而进一步提高用户体验。换句话说，随着核数的增加，假设在执行时间不变的情况下，工作负载的规模将逐渐增大，而这一价值的提升并非来自于更快的执行速度，而是来自于增加的功能。

Gustafson 定律的观点是：在 Amdahl 定律中，执行被分为串行和并行两部分。让我们先来看看单处理器的机器，它的执行时间是 $s+p$ ，其中 s 为串行部分的执行时间， p 为可并行部分的执行时间。现假设一个用户在一个拥有 P 个处理器的并行机上想要运行一个更大规模的工作负载，使得在新并行机上的执行时间与原工作负载在串行机上的执行时间相同，也即在并行机上运行新的、更大规模的工作负载的执行时间也是 $s+p$ 。我们估计在单处理器机器上运行这个更大规模的工作负载的执行时间是 $s+Pp$ ，隐含的假设是串行执行阶段的时间相同，且它与处理器核数无关。我们用 F 表示并行执行时间在总执行时间中所占的比例，即 $F = p/(s+p)$ ，则得出加速比的公式如下：

$$S_p = \frac{s + Pp}{s + p} = 1 - F + F \times P$$

(1.13)

图 1-16 给出了 Amdahl 定律和 Gustafson 定律基于 P 等于 100 的多处理器的比较情况，并给出了随着可并行执行工作负载占比增加的函数曲线。Amdahl 定律是残酷的，在 F 达到 90% 之

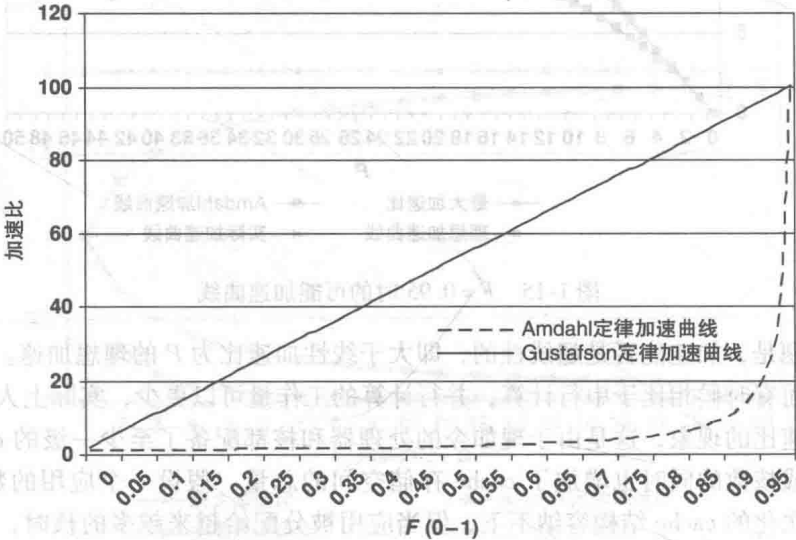


图 1-16 $P=100$ 时 Amdahl 定律和 Gustafson 定律的比较

前, 它的加速比是极小的, 当到达 90% 后, 它的加速比快速增长。另一方面, 在 Gustafson 定律中, 加速比随着 F 的变化是呈线性增长的, 因此在 Gustafson 模型中, 更多的工作负载可以利用拥有 100 个处理器的多处理器。Amdahl 模型将失败归结为并行架构的失败, 而 Gustafson 模型则更多地带给人们希望, 告诉人们多核架构的确是有益的!

1.5 技术挑战

过去, 在体系结构设计中唯一值得权衡的是成本 (即面积) 和时间 (即性能) 之间的关系。如今, 设计面临多个技术极限的挑战, 设计问题也主要是在多个技术约束下实现性能的最大化, 而其中一个重要的新约束就是功耗。CMOS 工艺已达到极限, 数字电路设计变得更加难以驾驭, 连线延迟、可靠性和设计的绝对复杂度都必须在设计过程中充分考虑。虽然一些问题可以在硬件 (电路) 层或软件层加以解决, 但体系结构多年来在维持 CMOS 技术可行性方面仍然起到了重要作用。

1.5.1 功耗和能量

现代微处理器的功耗主要由两大部分组成: 动态功耗和静态 (漏电) 功耗。

动态功耗

动态功耗是指门状态切换时所消耗的功耗。每个门的动态功耗是由该门输出电容充电和放电产生的, 每个时钟周期电路的平均动态功耗由下式给出:

$$P_{\text{dynamic}} = \alpha CV^2 f \quad (1.14)$$

动态功耗是由活跃因子 α (正比于门在一个周期切换的平均数)、门驱动的总电容量 C 、电源电压的平方 V^2 以及时钟频率 f 的乘积所得到的。这些影响因素的内在含义是很有用的, 动态功率正比于 αf , 因为它是电路中门状态切换的频率, 在电路中所有门状态切换的总能量消耗是 $CV^2/2$, 此能量用于门所驱动的电容的充电和放电。

在给定的电路中, 时钟频率随着电源电压的提高而提高, 当时钟频率降低时, 可以降低电源电压。事实上, 如果时钟频率 f 和电源电压 V 同时被修改, 电路消耗的动态功耗大致与时钟频率的三次幂成正比, 这一观测结果对微架构产生了深远的影响, 是对并行处理和并行架构今后走势的隐含驱动力, 而追求更快的时钟电路已不再是一个好的选择。

静态 (漏电) 功耗

动态功耗产生自电路状态的切换, 而静态功耗 (也称漏电功耗) 在每个电路中都一直存在, 无论电路的状态是否切换。

在 CMOS 中, 除非是门状态切换, 否则没有电流会通过门开关, 不幸的是, 晶体管不是完美的开关, 总有少量的电流会发生泄漏, 在 15 年前这个泄漏电流可以忽略不计, 但随着每代生产工艺阈值电压的降低, 泄漏越来越严重。阈值电压是晶体管关闭时所需的电压, 阈值漏电功耗依赖于阈值电压与温度, 具体关系如下:

$$P_{\text{subthreshold}} = VI_{\text{sub}} \propto Ve^{-k(V_T/T)}$$

漏电功耗随着阈值电压 (V_T) 的降低和温度 (T) 的升高呈指数级增长, 漏电功耗与电路面积和电源电压成正比, 但与时钟频率和电路的活跃程度无关。

近年来随着特征尺寸和物理门长度的降低, 阈下漏电功耗迅猛增长, 现已超越了动态功耗, 成为主要的功耗来源。就像动态功耗主要由处理器核消耗一样, 漏电功耗则主要是由 cache 消耗的。

1.5.2 可靠性

为了在晶体管沟道中保持电场强度恒定, 并避免破坏随着特征尺寸缩小而减小的连接, 每

一代生产工艺的电源电压都有所降低。此外,还有一种倾向是通过降低电源电压来进一步改善动态功耗和漏电功耗。存储在一个晶体管中的电荷量为 $Q = C \times V$, 该电荷量在每代新生产工艺中都会显著降低, 从而使得高速缓存和内核中的每一位存储都更容易产生瞬时故障。瞬时故障是电路故障, 此时电路虽然保持功能, 但它所包含的值却是损坏的。当存储单元(如 DRAM bit、SRAM bit、寄存器 bit 或 flip-flop)受到高能粒子撞击时(例如由宇宙射线发射出的中子粒子或由封装部件射出的 α 粒子), 就会产生损坏的值。为了防止这样的错误, DRAM 和 SRAM 通常有某种形式的误差检测和校正能力。

另一种类别的故障是由于芯片上环境变化所导致的临时故障, 例如由高温(热点)引起的错误可能会一直保持, 直到温度降低为止。由于热效应的维持时间的量级比周期时间大, 故障设备必须被禁用, 直到它的功能恢复正常。因此, 像瞬时故障、临时故障这类的故障是非破坏性的。

间歇性故障是由老化所引起的(即随着时间的推移出现的线路老化)。由于电路的行为恶化, 有可能在一定条件下出现间歇性故障。例如, 它可能不再满足时序要求, 这个问题可以通过增加周期时间或禁用发生故障的逻辑(提供备件或冗余)来解决。在这种情况下, 机器的操作被认为是安全失效(failsafe)的, 虽然性能降低, 但其功能是正常的。由老化引发的间歇性故障经常恶化为永久性故障。

片上多处理器可以利用丰富的线程冗余执行以提高其可靠性。例如, 两个冗余的计算可以相互检查, 并在一个发生瞬时错误时回滚到故障发生前的检验点, 或三个冗余线程可以比较它们的执行结果。此外, 在多核系统中故障核可以被禁用, 从而提供了一种天然的退化失效保护。

1.5.3 连线延迟

随着晶体管规模的减小, 其尺寸变得更小, 因此翻转速度变得更快。然而, 线路上信号的传播延迟并没有缩短, 线路传播延迟正比于它的 RC 常数, 线路的电阻正比于其长度和其横截面积的倒数。随着每代工艺的出现, 线路的横截面积不断缩小, 单位长度的电阻迅速增加。

通常, 微处理器芯片几个金属层之间存在过孔连接(即层与层之间的垂直连接)。不同的金属层具有不同类型、不同厚度的导线, 这些导线用于局部(层内)、中间(一个模块内)以及全局(整个芯片内)的通信。局部导线短、横截面薄, 而全局导线长、横截面厚。全局导线是分段的, 且使用缓冲器, 通过分割导线, 其延迟与导线长度呈线性相关而不是平方关系。导线也可以像逻辑一样进行流水化, 只要线路的长度缩短, 它的延迟就会减少, 但随着固定长度导线的增长, 延迟也在增长。

对于连线延迟而言, 流水级越深越好, 因为有意义的通信被限制在一个单一的流水线或两个流水级之间。然而, 在规定的时钟频率下, 其他结构的访问时间是由线路延迟控制的, 例如 cache、寄存器堆以及指令队列等, 除非它们的访问可以被流水化或它们的尺寸可以被缩小到在单位时钟周期内完成。连线延迟的影响对片上多处理器也是有利的, 因为 CMP 中的通信是分层的: 大多数通信发生在局部的每个处理器核内部, 而全局、核间通信被限制于核间数据和控制信息的交换。

1.5.4 设计复杂度

设计复杂度与片上门数量正以相同的速度不断增长, 即每两年翻一番。设计和设计验证的生产力也由于更好的 CAD 工具以及计算机平台计算能力的快速扩张而不断增长。然而, 设计

验证工程师的生产力仍远远落后于验证任务的复杂度，如今验证的成本和专门设计验证的工程师数量都在以极快的速度增长，在设计一款新的芯片的过程中，验证占据了主要的成本。无论是在门级、RTL 级（验证逻辑正确性）、核级（例如存储操作的正确性），还是在多核级（例如同步、通信、cache 一致性协议和存储一致性模型），都是需要验证的。

片上晶体管数量的增长已大大超过了设计和验证生产力的提升，同时由于动态功耗的限制，今天绝大多数的片上晶体管都致力于存储而不是计算。这些存储结构包括 cache、分支预测器、存储缓冲区、加载/存储队列、取指令队列、保留栈、重排序缓冲区和为保持 cache 一致性的目录结构。从设计验证的角度看，在芯片设计中，增加存储结构的规模要比合并新逻辑块简单得多，片上多处理器符合设计复杂度的发展趋势，因为多次复制相同的结构比设计一个大型复杂结构要简单得多。

设计验证已经成为一个主要的设计约束。体系结构的设计应易于验证，即“为验证而设计”，当两种设计相当时，易于验证的设计一定会更流行。

1.5.5 尺寸缩小极限和 CMOS 终点

CMOS 工艺已经快达到其尺寸缩小的极限，这就是所谓的 CMOS 终点，此时 CMOS 将出现量子效应。如图 1-6a 所示，按照目前的趋势发展下去，到 2020 年特征尺寸将进入 10nm 甚至更小，然而，特征尺寸不是一个晶体管的实际尺寸，描述一代工艺的特征尺寸指的是两根金属导线之间距离的一半（称为半节距），而晶体管的尺寸更小。例如，当半节距是 30nm 时，门的长度实际是 15nm。30nm 时代将要来临，CMOS 晶体管大小将迅速达到原子间的距离范围。

图 1-17 显示了 2007 年 ITRS 对 CMOS 特征尺寸发展趋势的预测。物理门长度大约是半节距的一半，2015 年，门长度达到 10nm，原子的半径为 $1 \sim 2 \text{ \AA}$ ($1 \text{ \AA} = 0.1 \text{ nm}$)，因此，10nm 的门长度是一个原子大小的 50 ~ 100 倍。此时，晶体管沟道中离散粒子数量的影响将变得十分明显，并将以十分复杂的方式影响晶体管的行为，晶体管的行为会越来越缺乏确定性，其可能性也会越来越多。晶体管的临界尺寸越接近于原子尺寸，晶体管的特性越接近量子物理学领域的特性，即所有状态都是不确定的，二进制逻辑被概率性的状态所取代，目前还不清楚这些设备是否是完全可用的。

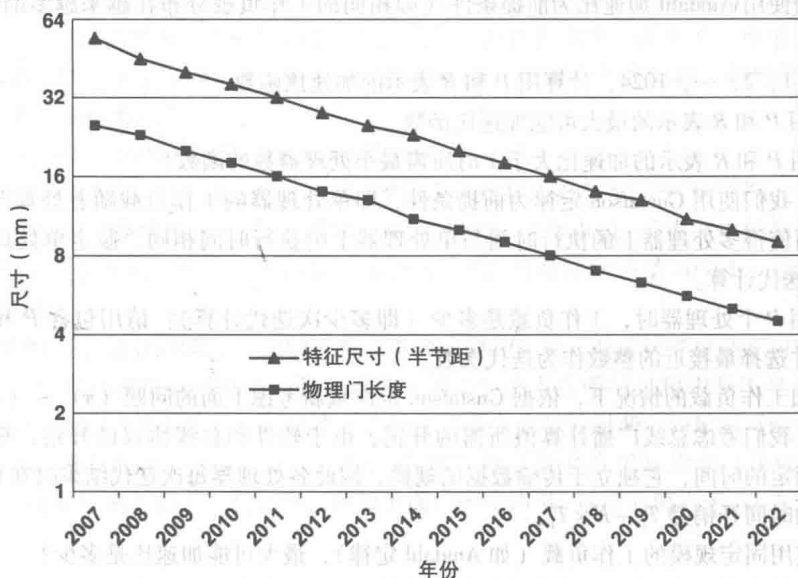


图 1-17 ITRS 对 2007—2022 年 CMOS 特征尺寸的预测

习题

1.1 现有一台基准机包含 load/store 指令集, 且浮点运算指令通过软件处理程序来实现, 现对其有两种改进方案。第一种改进方案是增加硬件浮点运算单元, 以加快浮点运算指令的执行, 据估计采用新硬件可以使浮点指令的执行时间变为原来的 $1/10$ 。第二个改进方案是增加更多的一级数据 cache 来加快 load/store 指令的执行, 据估计当使用与增加浮点单元消耗的面积来增加片上 cache 时, load 和 store 指令的执行速度相对于基准机可以提高 2 倍。

分别用 F_{fp} 和 F_{ls} 来表示浮点运算指令和 load/store 指令执行时间在总执行时间中所占的比例, 且这两类指令在同一时间不重叠执行。

(a) 当使用 Amdahl 加速比时, 若增加浮点运算单元带来的加速比大于增加 cache 空间带来的加速比, 则 F_{fp} 和 F_{ls} 之间的关系如何?

(b) 假设给定浮点运算指令和 load/store 指令所占的比例, 而不是 F_{fp} 和 F_{ls} , 且给出了浮点运算指令和 load/store 指令执行所需的平均时钟周期数, 基于这些数据, 是否能够算出哪一种改进更好? 能够使用 Amdahl 定律估计出两种改进的最大加速比吗? 若可以请给出计算结果, 否则给出理由。

(c) F_{fp} 和 F_{ls} 分别取何值时, 两种单独部署的改进都可以获得 50% 的速度提升?

(d) 现决定先部署浮点运算单元再来增加 cache 空间, 且在原工作负载中, F_{fp} 和 F_{ls} 的取值分别为 30% 和 20%, 则通过增加浮点运算单元能够获得的最大加速比是多少? 假定增加浮点运算单元实现了理论上最大的加速比, 那么相对于增加浮点运算单元后的性能, 增加 cache 空间所能获得的最大加速比是多少?

1.2 一个多处理器计算机有 1024 个处理器, 我们在这台机器上部署一种运算, 其中有 N 个迭代值需要计算, 然后各处理器之间需要进行数据交换, 即每次迭代后, 处理器在数据总线上广播计算值。每次迭代分两个阶段进行, 在第一个阶段中, 各处理器分配到 $k = N/P$ 个迭代计算, 其中 P 是计算所涉及的处理器个数; 在第二个阶段中, 各处理器将其计算结果逐个广播给其他处理器, 每个处理器需等待直到该通信阶段结束后才能开始下一个新的计算阶段。假定 T_c 是计算一次迭代所需的时间, T_b 是将计算结果广播到数据总线上所需的时间, 我们定义计算通信比 R 为 T_c/T_b , 注意当 $P=1$ 时, 不需要进行通信。

首先, 我们使用 Amdahl 加速比为前提条件 (即相同的工作负载分布在越来越多的处理器上), 在该条件下:

(a) 对 $K=1, 2, \dots, 1024$, 计算用 P 和 R 表示的加速比函数。

(b) 计算用 P 和 R 表示的最大可能加速比函数。

(c) 计算用 P 和 R 表示的加速比大于 1 时所需最小处理器数的函数。

然后, 我们使用 Gustafson 定律为前提条件, 即单处理器的工作负载随着处理器个数的增长而增长, 从而使得多处理器上的执行时间与单处理器上的执行时间相同。假定单处理器的工作负载是 1024 次迭代计算。

(d) 当使用 P 个处理器时, 工作负载是多少 (即多少次迭代计算)? 请用包含 P 和 R 的函数来表示, 并选择最接近的整数作为迭代次数。

(e) 在增加工作负载的情况下, 依据 Gustafson 定律重新考虑上面的问题 (a) ~ (c)。

最后, 我们考虑总线广播计算值所需的开销, 由于软件和总线协议的开销, 每个总线传输都需要一个固定的时间, 它独立于传输数据的规模, 因此各处理器每次迭代结束时在总线上广播 k 个迭代结果的时间开销是 $T_0 + K \times T_b$ 。

(f) 假设使用固定规模的工作负载 (如 Amdahl 定律), 最大可能加速比是多少?

(g) 假设工作负载规模增加 (如 Gustafson 定律), 最大可能加速比是多少?

1.3 本题是关于报告一组程序或基准测试集平均加速比的, 我们考虑 4 种报告多个程序平均加速比的

方法:

- 取平均执行时间的比例, S_1
- 取加速比的算术平均值, S_2
- 取加速比的调和平均值, S_3
- 取加速比的几何平均值, S_4

考虑习题 1.1 中基于基准机的两种改进方案, 一种是增加浮点性能, 另一种是增加存储性能, 现通过三个程序对其进行模拟: 一个没有浮点运算 (程序 1), 一个由浮点运算占主导地位 (程序 2), 最后一个在存储访问和浮点运算之间保持一定程度的平衡 (程序 3)。每个程序在三台机器上的执行时间由表 1-4 给出。

表 1-4 三个程序的执行时间

机器	程序 1	程序 2	程序 3
基准机	1s	10ms	10s
基准机上增加浮点单元	1s	2ms	6s
基准机上增加 cache 存储	0.7s	9ms	8s

- (a) 计算各改进方案相对于基准机的平均加速比, 如果仅单独考虑各平均加速比, 哪个改进方案是最好的?
- (b) 为了去除由于执行时间不同所带来的偏差, 我们首先将基准机的执行时间规格化为 1, 表 1-5 列出了规格化的执行时间, 分别计算两种改进方案相对于基准机的 4 类平均加速比, 如果仅单独考虑各平均加速比, 哪个改进方案是最好的?

表 1-5 三个程序的规格化执行时间

机器	程序 1	程序 2	程序 3
基准机	1	1	1
基准机上增加浮点单元	1	0.2	0.6
基准机上增加 cache 存储	0.7	0.9	0.8

1.4 在一台主频为 100MHz 的机器 M1 上, 观测到整数基准测试中 20% 的计算时间都花在了 Multiply (A, B, C) 子程序上, 这个子程序完成了整数 A 和 B 的相乘, 并将乘积赋给 C。此外, 每次调用执行 Multiply 子程序需要 800 个时钟周期。为提高程序的执行速度, 提出了一种用于在整数基准测试中提高机器性能的新指令 MULT, 如果有足够的数

据, 请回答下列问题, 如果没有足够的数

据, 仅回答“没有足够的数

据”。

- (a) 在这组程序中, Multiply 子程序执行了多少次?
- (b) M2 是一种实现了 MULT 指令的新机器, MULT 执行乘法需要 40 个时钟周期 (相对于 M1 执行需 800 个时钟周期的改进), 除了乘法, 所有不包含在 Multiply 子程序中的指令在机器 M1 和 M2 上的 CPI 都是相同的, 由于增加了复杂性, M2 的时钟频率是 80MHz, 请问 M1 比 M2 快 (慢) 多少?
- (c) 在机器 M3 上, 设计并模拟了 MULT 指令更快的硬件实现, 时钟频率也是 80MHz。据观测相对于 M1, M3 的速度提高了 10%, 请问这有可能么? 或者在模拟过程中是不是存在错误? 如果可能的话, MULT 指令在这个新机器上执行需要多少个时钟周期? 如果不可能, 请说明原因。

1.5 本题我们比较两种条件分支指令。首先, 我们看 MIPS 指令集中的 BEQ 指令 (详细资料请参考第 3 章), 该指令通常与 SLT 指令一起使用, 例如下面代码中当 R1 存储的值小于 R2 存储的值时, 产生分支。

SLT R3,R1,R2 /如果R1<R2, 则 R3设置为1, 否则R3设置为0

BNEZ R3,target /当R3中的值不为0时, 跳转到目标位置

此时需要两条指令。此外，我们也可以使用既能测试条件又能执行分支跳转的分支指令，这种方法可以节省一个算术/逻辑指令，但它也更复杂，因此执行周期更长，例如：

```
BLT R1,R2,target    /如果R2<R1，跳转到目标位置
```

有些人建议类似 BLT 的指令（如 BGE、BGT 等）应当被添加到 MIPS 指令集中。

(a) 首先考虑只包含 BEQZ 和 BNEZ 的基准机，请根据表 1-6 列出的动态指令组合比例计算平均 CPI。

表 1-6 基准机中的动态指令组合

指令	出现频率	执行周期数
算术/逻辑指令	40%	1
取数指令	25%	2
存数指令	10%	1
分支指令（不跳转）	8%	1
分支指令（跳转）	12%	3
其他指令	5%	1

(b) 添加 BLT 型分支指令后，时钟周期提高了 5%，但编译器能够去除与分支相关的 50% 的 SLT 指令，请计算新的平均 CPI。

(c) 请问添加 BLT 型指令是否是一个好主意？

1.6 假定基准微处理器以如下方式进行改进：

- (1) 为构建一个 16 路的 CMP，处理器核被复制了 16 次。
- (2) 每个处理器核添加一个浮点协处理器，这个协处理器使得每个核的浮点运算速度都提升了 4 倍，当浮点协处理器活跃时，主核处于空闲状态，且协处理器的存在并不影响非浮点运算指令的执行。

在新机器上，观测到如下现象：

- (1) 在每个核的执行时间中，有 30% 的时间用在浮点协处理器上。
- (2) 在总执行过程的 25% 的时间里，只有一个处理器核活跃，在其他的 75% 的时间里，有 4 个核活跃。

请问与基准机相比，新机器的加速比是多少？

工艺及其影响

2.1 概述

工艺在计算机系统结构的演变中一直扮演着非常重要的角色。工艺的进步促进了芯片结构的快速革新,本章中将分析由性能、功耗和可靠性这三个方面的需求驱动带来的结构革新。过去,体系结构被性价比左右,数个著名的结构设计得益于某些工艺参数的不均衡发展。比如,片上缓存(cache)就是当处理器速度远超过主存(main memory)读写速度时的产物;近年来,功耗则成为架构设计的主要约束。从微处理器发明以来,芯片面积和晶体管密度不断增加,使得时钟频率得以呈指数倍提升,甚至实现了更为复杂的电路设计。然而,当芯片供电电压接近下限,且功耗问题令人担忧之时,为了控制功耗增长,芯片结构的设计方法从单个高频处理器结构转变为片上多处理器结构。这种从单处理器到多处理器的微结构转变是由工艺进步引发的变革。同时,处理器的可靠性一直是高端服务器系统的关键问题。由于晶体管的特征尺寸随着时间推移而不断缩小,它们对瞬时故障也更加敏感。因此引入抗辐射设计来保护计算机系统免受由单粒子翻转引起的软错误影响。

这些例子说明工艺对计算机设计带来的影响,了解基本的工艺参数和特征以及每代工艺的更新对读者来说非常重要。这些知识可以帮助读者更好地领会在这本书的余下部分对结构设计的深入讨论。此外,在未来,计算机设计者将应对更多的由于片上特征尺寸不断缩小而导致的更加严峻的工艺挑战。同时,这些挑战也给计算机体系结构设计带来新的创新机遇。这些挑战将需要在各级软件/硬件结构中有重大突破,包括系统和应用层软件、体系结构、电路和材料科学等。

在这一章里,我们从介绍一个晶体管如何工作开始。这部分可以作为有MOS知识背景的读者的复习资料,也可作为不熟悉电气工程的读者的初级读物。一旦建立起来对CMOS门的工作原理和参数的认识,我们就可以描述每次晶体管工艺尺寸变更下的物理和电学特性。物理尺寸显著影响晶体管的电气特性。器件尺寸变化的明显优势是每次更新换代时晶体管密度增加,为计算机设计师提供越来越多的可用电路。如果电源电压与特征尺寸一样按比例缩小,那么每个晶体管的动态功耗降低,或者每单位面积的动态功耗在换代过程中保持恒定。动态功耗是电路从一个状态转换到另一种状态时所发生的能量开销。除了动态功耗,由于电路漏电导致的静态功耗也随着工艺变更迅速增长。不论状态是否切换,电路都会有漏电流。静态功耗与电路所占用的面积成正比。在本章中,我们解释了动态功耗和静态功耗的基本方程及其整个耗电过程对电路产生的性能影响。功耗问题已成为近年来设计师要面临的主要挑战之一。因此,每单位功耗所能提供的性能成为评判系统结构创新的一个新的衡量标准。动态功耗可以通过并行或流水操作以及关掉不使用的模块来减少。同样,供电电压对动态功耗也有着最显著影响。这些电源管理方法都将在本章阐述。

本章的第二部分探讨了一个新兴的设计问题,即极小尺寸器件的可靠性问题。由于晶体管尺寸缩小到了比用来蚀刻电路的光波还短,制造不精确正变得愈发常见。制造不精确导致芯片内(within-die)和芯片间(die-die)的电路特性差异明显。此外由于物理尺寸缩小到纳米范围,当电流密度极高时,器件老化速度极高。制造不精确和快速老化导致多种形态的设备故

障。在本章中,我们介绍几种突出的失效机制:瞬时故障、负偏置温度不稳定性(NBTI)、电迁移效应(EM)和时间依赖性的介质击穿(TDDB)。

本章涉及的主题有:

- 2.2 节和 2.3 节主要介绍电学基本定律、MOS 晶体管和 CMOS 门电路的特点。
- 2.4 节主要介绍集成电路工艺尺寸进步对电路性能的影响。
- 2.5 节主要介绍功耗和能耗,包括动态和静态功耗方程,在电路层和结构层降低功耗的技术,功耗和能耗相关的度量。
- 2.6 节主要介绍可靠性概念,包括错误的分类——可检测不可恢复错误(DUE),静默数据损坏(SDC),瞬时错误及其处理方法,错误保护码,可靠性度量方法,间歇性故障的物理原因(EM, NBTI 和 TDDB),永久性故障,工艺变化的影响等。

2.2 电学基本定律

电流是由动态(流动)或静态(存储)电荷(charge, Q)产生的。电荷有正负两种,负电荷是由电子的积累形成的,正电荷是由于电子缺失形成的。相同极性的电荷(+和+, -和-)相互排斥,极性相反的电荷(+和-)互相吸引。电荷可以存储在电容器或电池中。

一个电荷在其周围产生电场(E)。电场会向周围的电荷产生作用力,并可以将电荷从一个点移动到另一个点,从而产生电流(I)。电场方向是电场中正电荷移动的方向。电流是由电场驱动电荷的流动形成的。电流的单位是安培(ampere, A)。电流可以在导体(conductor)内自由流动,例如金属。大多数固体是绝缘体(insulator),绝缘体能在巨大的电场中不产生任何电荷或电流。半导体(semiconductor),例如硅,是独特的,因为它们可以有时表现出导体的特性,有时表现出绝缘体的特性。电荷和电流之间的关系由下式给出:

$$dQ = I(t) dt \quad (2.1)$$

当电荷受到电场的作用时,它就具有能量,称为电势。电势是电场的积分,两点之间的电势差被称为电压(voltage, V)。因而电压是每单位电荷的势能的差。电势和电压的测量单位均为伏特(volt, V)。

每当电流 I 在电压 V 的作用下流过两点之间,都会产生功耗。功耗的计算公式是 $P = VI$ (即电荷流过电位差的量)。功耗的单位是瓦特(watt, W)。能是功耗经过一段时间的积分。

2.2.1 欧姆定律

欧姆定律涉及电流和电压。在它的一般形式中,欧姆定律说明电压和整个负载电流成正比:

$$V = ZI \quad (2.2)$$

其中 Z 为负载阻抗(impedance)。电阻和电容是阻抗的两个例子。

2.2.2 电阻

如果负载为纯电阻,则阻抗称为电阻(resistance)。该电阻取决于构成该负载材料的电导率(或电阻率):

$$Z = R = \rho \frac{L}{WH} \quad (2.3)$$

其中 L 是导体的长度, WH 是在导体(宽×高)的截面积, ρ 是电阻率。在 VLSI 电路中线是主要的电阻性的电路元件。

电阻的功耗由下式给出:

$$P = VI = RI^2 \quad (2.4)$$

2.2.3 电容

电容是一种可以存储和保持电荷的器件。它是由两个电极板组成，两个电极板之间用被称作电介质（dielectric）的绝缘体隔开。电荷正比于施加到电极板上的电压：

$$Q = CV \quad (2.5)$$

其中 C 是电容。电容的通式是：

$$C = \kappa \epsilon_0 \frac{A}{d} \quad (2.6)$$

其中： A 为极板的面积， d 为平板间的距离， ϵ_0 是在真空或自由空间的介电常数， κ 是两板之间的绝缘体的相对介电常数或介电常数。二氧化硅（CMOS 中的电容器所使用的材料）的介电常数一般为 $\kappa=3.9$ 。

在电压为 V 的条件下，电容 C 存储的能量为：

$$E = \frac{1}{2} CV^2 \quad (2.7)$$

在 VLSI 电路中，动态功耗大多花在与所有组件（晶体管和线路）有关的电容充电和放电上。

当施加到电容上的电压值发生改变时，存储在电容器上的电荷不能瞬时改变，因为必须由电流从电源流至电容器来改变它。这个电流流经一个（可能是寄生）电阻器（电阻值 R ）。如果没有电阻，电容会瞬间充电。因为与电容串联的电阻器的影响，在电容器中存储的电荷数将与时间常数 RC 按指数变化。电容组件的电压不会瞬间改变，而是以指数变化，具有陡峭的初始斜率，接着渐渐变缓。该 RC 延迟（不是光速）在 VLSI 电路中决定了延迟和切换速度。

每一个电路元件都有与之关联的一些电容。例如，导线具有电容，门电路有输入和输出电容。在两条接近的电线之间也存在着交叉电容。

2.3 MOSFET 晶体管和 CMOS 反相器

场效应晶体管（FET）是电子开关，它具有三个主要的输入：栅极，源极和漏极。栅极和源极之间的电压可以控制源极和栅极之间是否有电流流动。金属氧化物硅场效应晶体管（MOSFET）是使用电容器将控制栅极的电压转移的 FET。

要了解 MOSFET 首先必须了解由电容控制器件构成的基本的 MOS 器件。这种电容器的电介质（绝缘体）是由二氧化硅（ SiO_2 ）构成的。电容器的两个极板一侧是由多晶硅（导体）栅极制成，另一侧是衬底，如图 2-1a 所示。衬底由掺杂硅制成。掺杂是通过向硅内注入（扩散）掺杂剂形成的。这两种类型的掺杂剂分别是 p 型和 n 型。在 p 型衬底上的多数载流子是正的。正的载流子是吸引电子的空穴。在 n 型衬底上的多数载流子是负的（电子）。少数载流子在衬底的数量比多数载流子的数量少几个数量级。

图 2-1 中，b、c 和 d 表明了栅极电压如何控制载流子在 p 型衬底中的分布。当栅极电压为负时，栅极充满了负电荷（电子），从衬底吸引大量（正的）载流子（图 2-1b）。如图 2-1c 所示，栅极电压略大于零，此时栅极带些许正电荷。栅极电荷排斥了在衬底上的大多数（正的）载流子，从而形成栅下的耗尽区（depletion）。当栅极电压进一步增大，大于阈值电压（threshold voltage, V_{th} ）时，少数载流子（电子）在衬底接近栅极附近聚集，而多数载流子被排斥到远离栅极的方向。电子在栅极下形成一层薄的反转层（inversion），它可以作为电子导电沟道。实际上，栅极的材料变为 n 型而不是 p 型。反转层的厚度和电导率随栅电压的增加而增加。

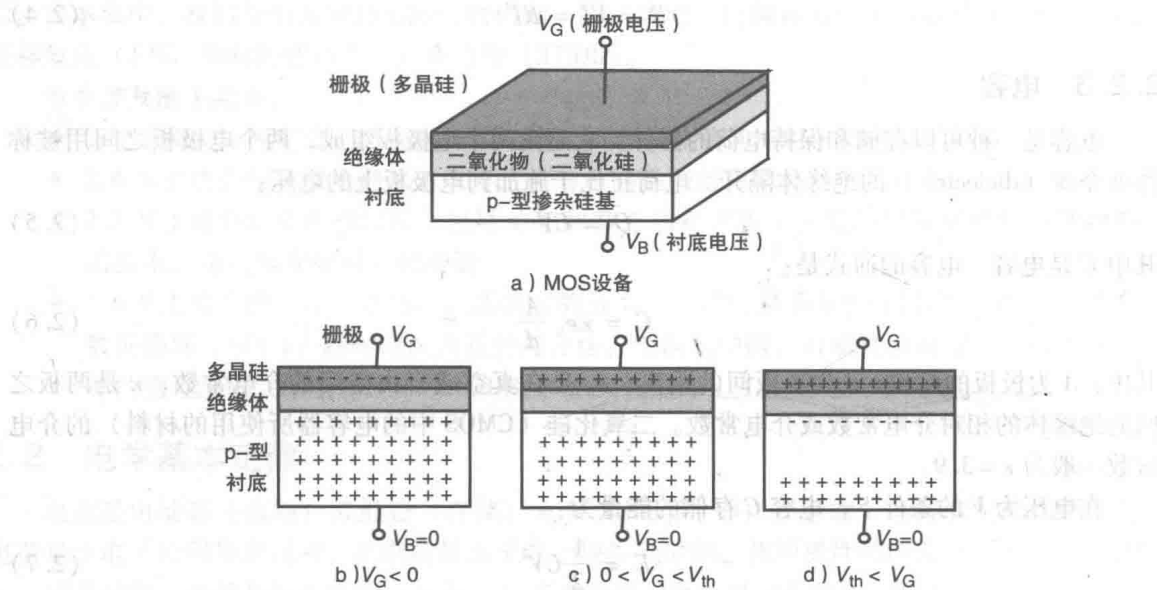


图 2-1 由栅极电压控制的 MOS 设备

在 n 型衬底中也可以观察到类似的行为。在这种情况下，多数载流子是电子，栅极电压为负，从而形成耗尽区。一旦栅极电压小于（负）阈值电压时，由邻近栅极的正电荷作为一个导电沟道形成反转层。从现在开始，我们专注介绍 nMOS 晶体管，即衬底为 p 型的晶体管。

通过 nMOS 晶体管的电流是由栅极电压控制的，栅极电压可以穿过氧化层决定导电沟道的厚度，如图 2-2 所示。两个被称为源极和漏极的重掺杂 n 型区放置在 MOS 器件的 p 型衬底的两侧，如图 2-1a 所示。

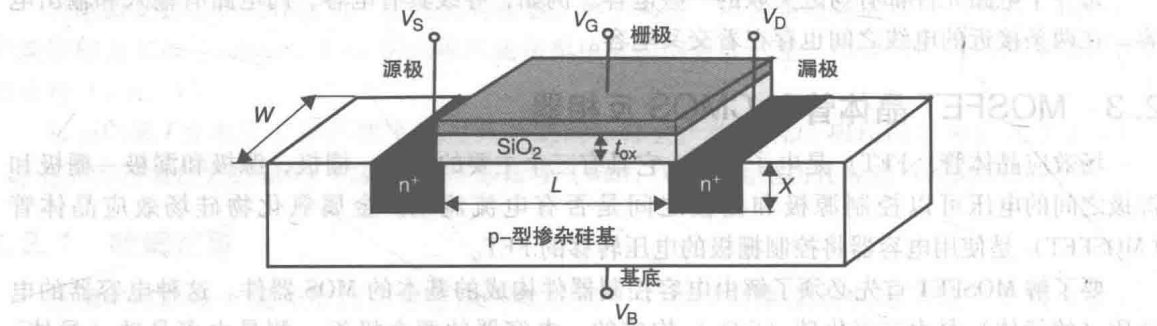


图 2-2 nMOS MOSFET

假设源极和衬底都接地。当 V_{GS} （即 $V_G - V_S$ ）为负时，源极和漏极之间的空间被正的多数载流子填充。没有电流在源极和漏极之间流动，因为源极产生电子和两磁极之间的多数载流子都是正电荷。然而，如图 2-1d 所示，如果 $V_{GS} > V_{th}$ ，少数载流子被吸引到栅极，在栅极下由电子形成反转层，在源极和漏极之间形成由少数载流子（即电子）构成的导电沟道。

当 $V_{GS} > V_{th}$ ，在漏极和源极之间的正电位在沟道产生纵向电场，其中电子从源极移动到漏极（即电流从漏极流向源极）。沟道成为栅极电压控制下的阻抗路径。栅极电压越高，则电子层越深，沟道的电阻越小。在源极和漏极之间施加一个正电压 V_{DS} ，在它们之间就会产生电流。电流大小与 V_{DS} 成比例。

当漏极接地时沟道如图 2-3a 所示。当漏极至源极电压增加时，漏极和源极之间的电流 I_{DS} 线性增加，但同时，栅极到漏极电压 V_{GD} 减小（ $V_{GD} = V_{GS} - V_{DS}$ ）。在某一时刻， V_{GD} 变得小于阈

值电压。这时，沟道在漏极被“夹断”。若 V_{DS} 进一步增大，夹断点会向源极移动，并且沟道从漏极后退，如图 2-3b 所示。跨过沟道从夹断点到源极的电压，在 V_{DS} 增加时 $V_{DS,sat} = V_{GS} - V_{th}$ ，并且电流 $I_{DS,sat}$ 保持恒定，不依赖于漏极到源极的电压。

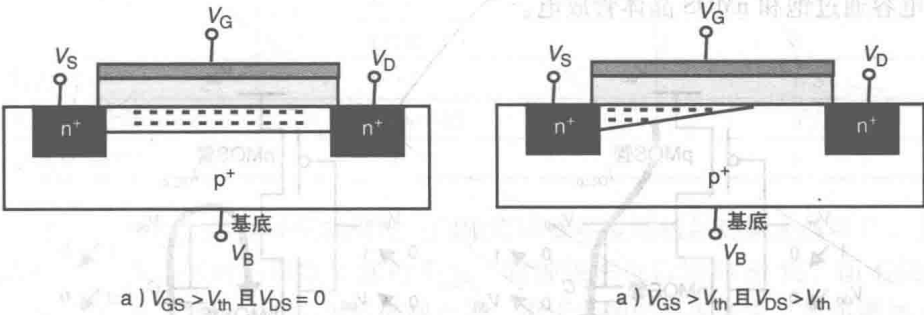


图 2-3 栅极电压和漏极电压在沟道的作用

总之，MOSFET 晶体管具有三个操作区域：

- 截止（阈下）区，没有电流可以在漏极和源极之间流动 ($V_{GS} < V_{th}$)。
- 线性区，其中 I_{DS} 随着 V_{DS} 线性增加 ($V_{GS} > V_{th}$ 且 $V_{GS} - V_{DS} > V_{th}$)。
- 饱和区，其中 I_{DS} 保持不变 ($V_{GS} > V_{th}$ 且 $V_{GS} - V_{DS} < V_{th}$)。

在线性区，当 V_{DS} 小于 V_{GS} 时，漏极和源极之间的电流由下式给出

$$I_{DS} \approx \beta(V_{GS} - V_{th})V_{DS} \tag{2.8}$$

其中 $\beta = \mu C_{ox}(W/L)$ 被称为晶体管的增益， μ 是电子的迁移率。在线性区中，沟道中的电流与沟道两端的电压成正比，并且比例因子取决于栅极到源极的电压。因此，在该线性区中，栅极扮演着阻抗的角色，其阻值由栅极到源极电压的控制。

当 V_{DS} 达到 $V_{DS,sat}$ 时沟道被夹断，并在整个沟道的电压保持在 $V_{DS,sat} = V_{GS} - V_{th}$ 。饱和电流由下式给出：

$$I_{DS,sat} \approx \frac{\beta}{2}(V_{GS} - V_{th})^2 \tag{2.9}$$

因为 V_{GS} 是 0V 或 V_{dd} （其中， V_{dd} 是电源电压），所以上述饱和电流的方程为变为

$$I_{DS,sat} \approx \frac{\beta}{2}(V_{dd} - V_{th})^2 \tag{2.10}$$

在饱和模式下，晶体管充当漏极到源极之间的电流源，并且与栅极到源极之间的电压的二次方成正比。

pMOS 晶体管具有类似的特性，不同之处在于基底是 n 型（多数载流子是电子）。这需要负栅极电压低于（负）阈值电压，形成一个正电荷沟道。

nMOS 和 pMOS 晶体管可作为通过栅极电压控制的开关。在 nMOS 晶体管，当栅极到源极电压小于阈值电压时，不论 V_{DS} 的值为多大，在漏极和源极之间都没有电流流动，此时衬底充当了绝缘体。当栅源电压比阈值电压高时，电流可以在漏极和源极之间流动。这个电流的强度取决于栅源电压。

问题在于，当开关导通时，电流流过它，一个 nMOS 器件的功耗保持在稳定状态。为构建出低功耗逻辑门器件 MOS 晶体管，将 nMOS 和 pMOS 晶体管相结合，这种方法被称为互补型 MOS（complementary MOS）（或 CMOS）。

图 2-4 显示了一个基本的 CMOS 反相器的工作原理。当输入为高电平时（参见图 2-4a），pMOS 晶体管关断而 nMOS 晶体管导通。反相器内没有电流流过。作为输入切换到 0 时，pMOS

晶体管导通并饱和，电流流过它为输出电容充电，而 nMOS 晶体管关断。在这一过程中，输出电压上升到 V_{dd} 。一旦输出电容进行充电时，pMOS 晶体管导通，nMOS 晶体管被关断，流经反相器电流消失。输入从低切换到高是一个和上述过程相对称的过程，如图 2-4b 所示。在这种情况下，电容通过饱和 nMOS 晶体管放电。

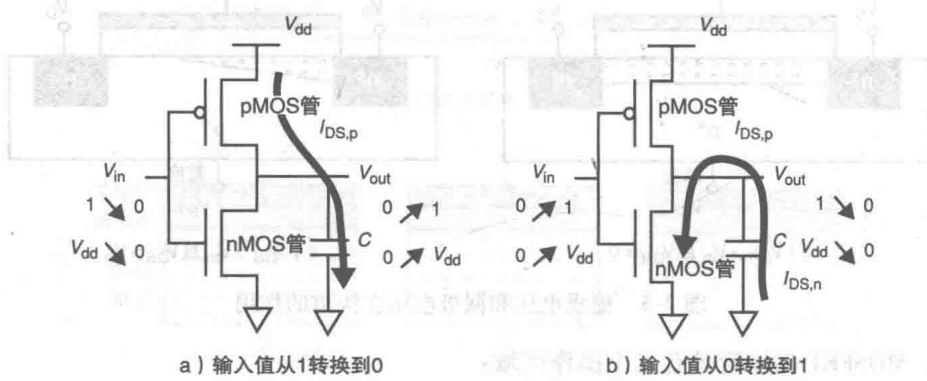


图 2-4 CMOS 反相器

2.4 工艺变更

随着技术的不断进步，每一代工艺每 2~3 年都会以缩减因子 $S = \sqrt{2}$ （约 30%）减小特征尺寸和电压，每 4~6 年 $S = 2$ （即 50%）减小工艺尺寸。参看图 2-2，表 2-1 中所示的尺寸和电压等参数都进行了 $1/S$ 的缩放。

表 2-1 电压和空间维度的变更技术

特点/电压	变量	变更范围
沟道长度	L	$1/S$
沟道宽度	W	$1/S$
栅极氧化层厚度	t_{ox}	$1/S$
接合深度	X	$1/S$
供电电压	V_{dd}	$1/S$
阈值电压	V_{th}	$1/S$
线规模（宽，间距，高）	w, s, h	$1/S$

表 2-2 所示为器件和连线的由工艺变更所改变的特征参数。第 2 列表示各个特性和变更参数之间的比例关系。在每一代工艺变更中，按比例变更电压和空间尺寸，电路密度增加了 2 倍，而性能（时钟频率）增加了 41%。若遵循表 2-1 的比例规则，整个器件电场保持不变，从而避免设备故障和消除大部分非线性效应。此外，每个门的动态功耗（VI）以 S^2 倍数减小，从而动态功耗密度保持恒定。然而，我们将看到，阈值电压按比例缩小对漏电功耗的增长有着指数级的影响。

表 2-2 设备和线特征

器件/线的特征	相关原则	变更范围
晶体管增益 (β)	$W/(L \cdot t_{ox})$	S
电流 (I_{ds})	$\beta (V_{dd} - V_{th})^2$	$1/S$
阻抗	V_{dd}/I_{ds}	1

电容的充电或放电。每当栅极输入从1到0变化时（见图2-4a），电流必须从 V_{dd} 流过负载流以移动电荷。然而，一旦栅极状态转换完毕，就不会再产生动态功耗。

动态功耗方程

功耗是电流和电压的乘积，并且在一段时间 T 内消散的能量是功耗的积分。假定时间 T 是对输出电容负载充电的时间。由充电耗散的能量可根据下式给出：

$$E = \int_{t=0}^T P(t) dt = \int_{t=0}^T v(t) i dt = C \int_{v=0}^{V_{dd}} v(t) dv = \frac{1}{2} CV_{dd}^2 \quad (2.11)$$

设 f 是时钟频率。在一个周期内时钟信号切换两次状态。但是，一般情况下，门在一个时钟周期期间转换状态至多一次。一个门消耗的动态功耗的总平均功耗（每单位时间的能量）的公式由下式给出：

$$P_{dynamic} = \alpha CV_{dd}^2 f \quad (2.12)$$

门的活动因子 α 考虑了栅极转换状态时的时钟周期。

如果 C 是电路中所有门电路的电容负载和，并且 α 是在电路中的门的平均活性因子，则式(2.12)可以解释为在整个电路动态功耗。

根据表2-1和表2-2，电源电压和晶体管的输入电容都变更 $1/S$ 倍，同时频率在每一工艺节点都变更了 S 倍。因此，一个给定的电路的动态功耗减小了 S^2 倍（根据实践，系数为2）。由于集成在一个特定区域的电路的数量变更了 S^2 倍，所以功耗密度在整个过程中保持不变。这是非常令人满意的，因为它意味着，动态功耗不会妨碍集成。然而，为了实现这个理想情况，电压必须缩小为 $1/S$ 倍。如果电源电压不按技术节点的速率缩小 $1/S$ 倍（用以驱动频率），则在单位面积的动态功耗不会保持恒定，而会增加。此外，如果该芯片的尺寸增大，则动态功耗也会成比例地增长。在过去的几年中，电源电压并没有按我们理想的比例缩小，在表2-1给出了理想比例（ $1/S$ ）规则。这个趋势已导致了整个芯片的功耗的显著增加，并刺激了针对功耗的体系结构设计。

注意，在门状态转换的短暂时间内（栅极电压的上升和下降期间），pMOS和nMOS晶体管同时被导通，从而 V_{dd} 和接地端之间产生了暂时的电流，并且在栅极转换状态时产生了动态功耗。与对输出电容充电和放电所需的功耗相比，这个寄生功耗通常被认为是可以忽略不计的。

低功耗设计技术

对于给定的工艺，有几种方法可以用来减少数字电路设计的功耗。事实上，动态功耗（式(2.12)）暗示了4种方法，以减少给定电路的动态功耗，即降低 α 、 C 、 V_{dd} 或 f 。降低 α 需要电路级技术来降低完成给定功能的逻辑活动。为了避免伪栅极和寄存器值的转换，整个电路的电源可以在电路不使用的周期内被切断（例如，一个浮点单元可能在执行整数程序的同时被禁用）。这就是所谓的电源门控，即一个单元当它在不使用时就断开其电源，如图2-6a所示。

在一个芯片中最活跃、最耗电的网络是时钟网络，因为时钟信号在每个时钟周期切换两次。时钟门控是指当电路不使用时，断开其时钟，如图2-6b所示。时钟门控降低了由时钟网络和选通电路消耗的动态功耗。条件选通是指，如果存储在寄存器中的值是不变的，那么到达寄存器的时钟信号被禁止，也降低了因电路的动态功耗。条件控制如图2-6c所示。在这个例子中，每当信号 c 处于活动状态时，寄存器B的值被存储到寄存器A中。在最顶层的设计中，A的下一个值在每一个时钟内被选择（基于 c 的值）和选通。在较低层的设计中（有条件的选通），只有当信号 c 是有效的，B中的值才会被选通到A。

时钟频率对动态功耗有线性影响，但降低了时钟频率对性能的直接负面影响。在实践中，

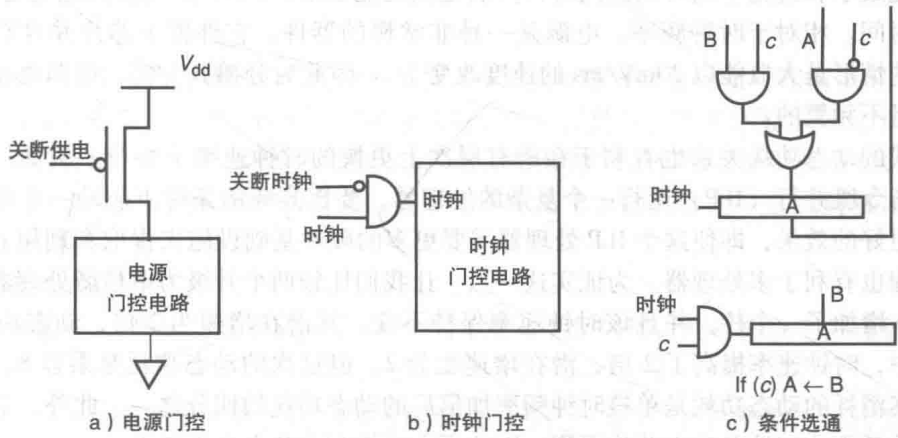


图 2-6 电路级的节能技术

可以减小非关键路径的部分电路的频率。

很明显，大幅降低动态功耗的最佳方式是降低供电电压 V_{dd} ，即名为电压缩放（voltage scaling）方式。效果是二次方的。问题是，对负载进行充电和放电的饱和电流 $I_{DS,sat}$ 与 $V_{dd} - V_{th}$ 的平方成正比（式（2.10））。 V_{dd} 不断减小使得保持相同的时钟频率变得不可能，因为信号的上升和下降时间不能满足栅极的噪声容限。为了避免这个问题，一个方法是减小 V_{th} 同时也减小 V_{dd} 。然而，这种方法也有局限。首先，我们将在 2.5.2 节看到，降低 V_{th} 时静态功耗呈指数增长。其次，栅极的噪声容限变小，栅极可能变得不可靠。多阈值 CMOS（MTCMOS）电路是解决这个难题的一种方法。在 MTCMOS 电路中，有两种类型的晶体管：high- V_{th} （低泄漏）和 low- V_{th} （高泄漏）。电路的设计为了速度而采用 low- V_{th} 晶体管并通过 high- V_{th} 晶体管连接到电源。在待机模式下，high- V_{th} 晶体管完全切断电路的电源（图 2-6a），从而消除待机漏电功耗。High- V_{th} （缓慢，低静态功耗）的晶体管可以在非关键路径上使用，而 low- V_{th} （快速，高的静态功率晶体管）在关键路径上使用。

降低电压和频率为体系结构设计提供了新的机遇。随着电压的降低，频率必须按比例减小。当电压和频率变化成比例，动态功耗是（大致）与频率的立方成比例。这一观察结果对计算机体系结构有着深刻影响。假设一个设备（如微处理器）运行在 1GHz 的频率下，并消耗 100W 的功耗。如果频率和电源电压都按比例缩小到原来的 1/10 时，设备现在运行在 100MHz 的频率下（性能下降到 1/10），但其动态功耗仅为 100μmW，这个功耗在手持设备的功耗范围内。

动态功耗方程的另一个应用是在不同功耗水平运行器件。必要时通过简单地调节电源电压和工作频率，设备可以在不同的性能/功耗点运行，在合理的性能损失中节省大量的功耗。这个调节可静态或动态进行。例如，一个完整的系统（例如笔记本电脑），当它被连接到一个电源时可在最大功耗下运行，而在电池供电时则运行在低功耗状态。不同功耗水平可以被应用到不同的活动水平，取决于其性能的关键因素。

一个更激进的策略是对每个有自己的电源电压的电路以不同的频率范围进行分区（如微处理器）。频率和电压可以在每个域中进行调整，以优化基于每个结构域的功能的整体的功耗/性能。频率和电源电压也可以在单个或多个频域动态地改变，从而在时间关键和非时间关键的阶段使用 DVFS（Dynamic Voltage and Frequency Scaling，动态电压和频率调整）运行。DVFS 是适用的，例如在实时系统中，处理器必须满足最后期限，并且可以为准时满足截止时间调节其频

率,在功耗最小和避免空闲时间的同时仍然满足其功能要求。DVFS 的主要问题是改变 V_{dd} 所需的等待时间。相对于时钟频率,电源是一种非常慢的器件,它外置于芯片并且有很大的惰性。当前的情形是大概能以 20mV/ms 的速度改变 V_{dd} 。因此为分摊其开销,电源电压的变化必须是很小且不频繁的。

立方式的动态功耗关系也有利于在所有层次上更快的时钟速率下并行。例如,在低频率下,利用指令级并行 (ILP) 运行一个复杂的处理器,要比高频的条件下驱动一个简单的五级流水线有更好的效果,即使这个 ILP 处理器需要更多的硬件基础设施来提取和利用并行性。动态功耗方程也有利于多处理器。为证实这一点,让我们比较两个升级为单核微处理器硬件:在升级 1 中,增加了一个核,并且该时钟速率保持不变。其潜在增速为 2 倍,动态功耗乘以 2。在升级 2 中,时钟速率提高了 2 倍,潜在增速也为 2,但这次的动态功耗是乘以 8。因此双核的解决方案消耗的动态功耗是单核时钟频率加倍后的动态功耗的四分之一。此外,双核解决方案的动态功耗更均匀地分布在芯片面积,因此不太可能在芯片上产生热点。

为了进一步说明这一观点,考虑重复执行的硬件功能(如指令执行的微处理器)。如图 2-7 所示,功耗可以通过将硬件功能流水化或通过并行处理得到降低。原电路的时钟周期为 T ,并有 16FO4 的延迟。输入值和输出值是以 $f=1/T$ 的速率进行处理的。如果该函数可以在两个阶段执行流水线,每个阶段具有 8FO4 逻辑延迟和相同的时钟速率 f ,然后输入和输出值仍然以相同的速率处理。然而,现在的电源电压可以减小一半,因为逻辑延迟(在 FO4 为单位)已经减半。因此,在整体性能相同的情况下,流水线执行的动态功耗是原始硬件动态功耗的 $1/4$ 。

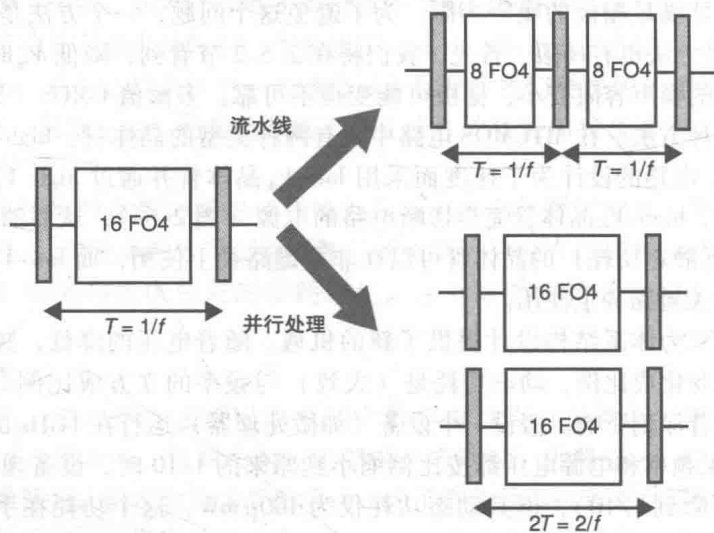


图 2-7 用流水和并行的方式降低动态功耗

若原设计不能流水,两个连续的输入值的处理也可以并行完成,如图 2-7 所示。该函数的硬件必须被复制,但是,由于两个平行单元并行工作,它们在主频为原电路一半的情况下可以达到相同的性能水平。在这个过程中,电压可以减少一半。因此,每个并联装置花费原来电路功耗的八分之一,总体而言,对于两个单元,动态功耗被降低到四分之一。

热点效应

除了平均功耗,芯片设计人员必须考虑空间或时间的峰值功耗,以避免极端的热效应。例如,一个高度活跃的阶段会导致芯片过热。有些程序阶段可能过度地使用某个功能单元模块,在芯片上形成局部热点。这种热点可能会导致单元模块发生故障甚至崩溃。这样的模块必须暂时关闭或节流使局部温度限制在可忍受范围内。

2.5.2 静态功耗

静态功耗是晶体管关断时的功耗。参见图 2-4，一旦栅极状态改变，在栅极应该没有功耗，因为跨越晶体管的电位是零，没有电流能够流过关断的晶体管。不幸的是，这是 CMOS 反相器的理想模型的特性。即使晶体管在关断状态，也存在多种原因会导致漏电流 (leakage current) 产生，这些电流在任何时候都产生静态功耗。漏电流最重要的来源之一是亚阈值漏电流 (sub-threshold leakage)。

亚阈值漏电流

当栅源电压 V_{GS} 降低且低于 V_{th} 时，电流 I_{DS} 并没有下降到零，而根据以下公式呈指数减小：

$$I_{DS_{sub}} \approx I_{DS0} e^{(V_{GS}-V_{th})/1.5v_T} (1 - e^{-V_{DS}/v_T}) \quad (2.13)$$

其中 v_T 是依赖于温度的热电压， v_T 与温度成反比（开氏温标下），在室温下大约为 25mV； I_{DS0} 是在阈值栅极电压时的电流。虽然由于亚阈值泄漏产生的电流很小，但会随着阈值电压的减小迅速增长。当 nMOS 晶体管为 OFF， $V_{GS}=0$ ， $V_{DS}=V_{dd}$ （比 v_T 大得多），亚阈值泄漏产生的静态功耗由下式给出：

$$P_{sub} = V_{dd} I_{DS_{sub}} = V_{dd} I_{DS0} e^{-V_{th}/1.5v_T} \quad (2.14)$$

其中 $I_{DS0} = \beta v_T^2 e^{1.8}$ 。

亚阈值漏电流与阈值电压具有直接的指数关系。随着阈值电压降低，亚阈值漏电功耗呈指数增长。漏电功耗与温度（开氏温标下）因为依赖于 v_T 也成倍增加。当通过一个因子为 $1/S$ 的变更工艺， $V_{dd} I_{DS_{sub}}$ 不会改变，因为 V_{dd} 缩放为 $1/S$ ，同时晶体管增益 β 变更为 S 倍。然而，阈值电压也变更为 S 倍，这将导致静态功耗的快速指数增加。因此，亚阈值功耗最近成为功耗的主要组成部分，堪比动态功耗，而这一趋势在未来工艺尺寸按比例缩小时势必更糟。

式 (2.14) 适用于整个器件密度均匀的电路。要获得整个电路的静态功耗，只需将 P_{sub} 乘以因子 A ， P_{sub} 正比于所占用的电路面积， A 是实际上在电路器件的数量。

载流子隧道效应（门级漏电流）

静态功耗还有其他来源。其中之一是由于穿过栅极电介质 (SiO_2) 的载体隧道形成的栅极漏电流。随着设备的尺寸缩小，栅绝缘体的厚度 (t_{ox}) 达到十几埃 ($1\text{\AA} = 0.1\text{nm}$)，达到了原子尺寸规模。介电材料是如此之薄，以至于载流子可以使用它们的方式（隧道）穿过它，仅仅通过它们自身的热能就可以做到。这导致了栅极电流泄漏，并与绝缘体的厚度呈指数下降。英特尔最近用二氧化铪 (HfO_2) 取代了二氧化硅。二氧化铪是一种 high- k （高介电常数）材料，可以制成 45nm 栅极电介质。电容正比于电介质的介电常数并反比于它的厚度（参照式 (2.6)）。

当栅极电介质具有更高的介电常数，它可以更厚并且在相同的电压下保持相同的电荷。晶体管增益 β 和切换速度都正比于栅极电容。因为隧道电流随着电介质厚度呈指数降低，high- k 介电材料，诸如 HfO_2 ，能非常有效地解决栅极泄漏问题，并且将可能取代二氧化硅作为在将来的 MOSFET 栅电容的介电材料。降低栅极漏电流的方法有可能属于材料工程研究的领域。

其他次要的电流泄漏的来源还有流过扩散区与衬底之间的电流。在本章的其余部分，我们忽略了除了亚阈值漏电流的所有其他形式的静态功耗。

降低静态功耗

适用于动态功耗的大部分技术对静态功耗也是有帮助的，但是它们的效果有时会不同。减少 V_{dd} 对静态功耗有线性影响，但减慢了电路速度，并影响其可靠性。最大限度地减少在电路设计中门和存储器单元的数量，这会对面积和静态功耗产生积极的影响。

如图 2-7 所示,在减少静态功耗方面,流水线是一个比并行执行更好的技术,因为流水线路与原来的电路占据大致相同的芯片面积,并因此静态功耗没有改变。相对而言,并行解决方案消耗了 2 倍的面积和静态功耗。从整体功耗来看,流水线的解决方案是最佳的,除非该函数不能被流水化。设计首先应尽可能部署流水线,然后再考虑并行。

功耗门控(也称为 gate- V_{dd} ,见图 2-6a)是非常有效的,因为它完全消除了电路的静态功耗(如一个未使用的单元)。问题是,一旦必须进行恢复,所有存储元件的内容就被破坏了。此外,性能和(动态的)功耗代价必须为设备供电做出牺牲。附加电路必须监视设备的开启和关闭的需求,并且必须切换打开和关闭电源。这些电路同样存在静态功耗。

根据式(2.14)可知,最关键的参数是阈值电压。基于临界性能的、拥有不同阈值电压的晶体管可以在整个电路中优化静态功耗。阈值电压甚至可以通过在源极和衬底之间应用一个反向偏置电压而动态地增加,这被称为动态 MTCMOS 方法。重新激活该电路,偏置电压被去除。电压门控和动态 MTCMOS 之间的区别在于被存储在存储单元中的数据能否被保护。这个动态调控可以在每个电路中有选择地完成。由于衬底上的 RC 常数是很高,在高和低阈值电压之间的转换需要的延迟和能量比门控 V_{dd} 要高得多,这也是动态 MTCMOS 的缺点。

片上缓存应用

在芯片设计中的多数晶体管主要用于存储,而不是用于计算。这些存储结构在现代的核中包括 cache、分支预测器(branch predictor)、存储缓冲器(store buffer)、加载/存储队列(load/store queue)、预取队列(instruction fetch queue)、发射队列(issue queue)和重排序缓冲器(re-order buffer)。由于每类存储的每比特的利用率随着结构尺寸的增加而下降,随机访问这些存储结构的概率也随之降低。静态功耗是由芯片上的每一个晶体管产生的,不论它是否被频繁使用,所以这些存储结构所产生的静态功耗占了总静态功耗的很大一部分,尽管它们的大部分都在很长时间内闲置。例如,一个大型的 L2 cache 行很少被访问,因为二级缓存上仅存取在 L1 cache miss 的访问,这些访问在二级缓存行中广泛分布。此外,cache 行的很大一部分是“死亡的”,即在从缓存中被删除之前它们将不会再次被访问。

为节省片上高速缓存漏电,可以在电路级切断各个高速缓存的电源(如使用电压门控或门控 V_{dd}),也可以通过增大阈值电压(动态 MTCMOS)将它们切换到休眠模式,或通过减少供给电压电平(reduced- V_{dd})来实现。当 cache 的电源被切断时,存储在其中的数据都将丢失。最新的数据副本必须在低层的存储中备份,并且下次访问此地址会出现 miss。这种方法的成功基于能够准确预测 cache 的“死亡”时间点,即该时间点后该 cache 地址永远不会被访问直到其被删除。除非该预测是成功的,否则这一方案增加了 miss 率,从而影响整体性能和动态功耗。另一种利用电压门控的方法是基于应用需求调整 cache 大小。这种方法主要是 cache 利用应用程序的不同,并在申请时动态改变应用程序的工作集大小。因此,通过关掉未使用的 cache 静态或动态改变 cache 的大小可以节省大量的静态功耗。静态调整将在执行整个应用程序前完成,而动态调整是在执行过程中根据需求的变化改变 cache 大小。编译器可以在早期丢弃“死亡”地址或调整 cache 大小。

一个不太激进的方法是让“cache 休眠”,通过增加 cache 中的晶体管的 source-to-body 的电压,这样会引起阈值电压的升高。然而,在休眠和非休眠模式之间进行切换时,cache 行性能和功率的代价很高,并且电路会很复杂。

休眠缓存的引入有希望限制静态功耗,同时不会造成过度的性能下降。休眠 cache 行可以是两种模式中的一种:(1)低泄漏休眠模式,其中数据将被保留,但不能被访问;(2)一种高泄漏休眠模式,在 cache 访问时使用。在休眠模式下的 cache 行的电源电压被降低到不会破坏数据的尽可能最小的电平。这个电平比阈值电压高。两种功耗电源(一个是标称的,一个是

用于休眠的电源电压) 必须达到所有的 cache 行, 每个 cache 行有一个休眠位来控制选择电源功耗模式。用启发式的方法决定不久的将来哪一行将会被访问, 并且这些行将保持在清醒状态。实际上, 休眠 cache 用动态电压缩放的方法来降低其漏电功耗。cache 访存时休眠唤醒的唯一牺牲就是很小的延迟和一点能量。小型的 L1 休眠 cache 的简单启示是将所有的 cache 行每隔几千个周期就转换到休眠模式下。休眠 cache 行被唤醒, 并在它的第一次访问中激活全部标称功率。它保持在满功率状态, 直到当前时间间隔结束时, 此时所有 cache 行被复位到休眠模式。根据这个简单的方案和访问一个休眠 cache 不超过一个周期的唤醒代价, 一级缓存的全部流量功耗可以降低 75%, 并且对性能的影响有限。

对于大型的 (可能共享的) L2 cache, 它的 cache 行很少被访问, 所有 cache 行都可以在长时间内保持在休眠的状态。当 L1 访问失效而访问 L2 的 cache 行时, 它先达到全功率, 然后进行访问。访问行之后, 它返回到休眠的状态。相比于 L1 cache 的多周期失效损失, 用来唤醒线路的增加的 L2 失效损失是可以忽略不计。

2.5.3 功耗和能量指标

当前, 功耗已成为主要的设计考虑, 在此之前, 面积和性能是公认的衡量 VLSI 设计质量的标准。设计的好坏是由“面积乘延迟” (AD) 来衡量的。对于给定的性能标准, 它的值越高说明设计越差。给定两个 AD 值相同的设计, 面积和时间之间可能有不同的折衷: 低速和小面积或高速和大面积。而面积仍然是一个问题, 片上硬件的指数级增长以及目前电源和功耗所带来的硬性限制, 使得这种担心变为次要的。因此, 类似的衡量标准已制订, 包含了功率、能量和性能。

功率是瞬时能量, 单位时间内的能量。功耗产生热量, 如果不以热量产生的速度将热量释放出去, 就会引起升温。一个体系结构的具体实施方案包括由它的散热和封装技术决定的功率范围。当功率超过这个范围时, 一个现代的处理器的必须具备一种机制来关闭或减缓自身速度, 以避免热效应的危害, 如影响性能, 导致暂时性的电路故障, 降低芯片的预期寿命, 甚至损坏芯片。频率越低, 功耗越低, 这意味着设计可以通过最大限度地发挥其执行时间来降低功率。因此, 虽然功率是一个重要的设计目标, 但不能被单独用作设计质量的度量。在实践中, 必须有执行时间的约束。

能量是环境中有限电源供应的一个重要指标, 如电池。能源是功率在一段时间 T 内的积分:

$$E = \int_{t=0}^T P(t) dt = \int_{t=0}^T P_{\text{dynamic}}(t) dt + \int_{t=0}^T P_{\text{static}}(t) dt = E_{\text{dynamic}} + E_{\text{static}} \quad (2.15)$$

能量是执行的任务一段时间消耗的总功率。如果 E 是能量, P 为平均功率, 而 D 是延迟 (执行时间), 则 $E = PD$ 。通常情况下, 更多的功率转化为更高的性能。因为动态功率是频率的超线性函数, 并且延迟反比于频率, 最大限度地减少动态功耗依然等同于执行时间的最大化。对于瞬间, 如果频率减半, 功耗 (P) 在理想的情况下减少八分之一。减少一半的频率会增加 2 倍的延迟 (D), 动态功耗下降 1/4 (由 $E = (1/8)P \times 2D$ 得)。然而, 静态能量随 D 而增加。因此, 降低频率和性能导致了能量消耗总量的上升, 因为静态功耗开始占据主导地位。

要提高性能在设计度量中的重要性, 延迟的影响必须在指标中加以强调。能量延迟积 (ED) 更强调性能, 适合表征工作站和桌面环境。最后, ED^2 (能量延迟平方积) 是一种用于测量高性能系统的标准, 如超级计算机或高端服务器, 其中的性能是最重要的并且能源供应和冷却能力很强大; ED^2 是相对于性能的一个非常积极的指标。随着频率的减小, 度量的动态分量保持不变或增大, 因为动态功耗和频率之间的关系顺序在现实中小于三次方。同时, 度量的

静态部分的增长为 D^2 。

最后对处理器而言一个重要指标是每条指令所花费的能量 (EPI)。它等于花费在任务的总能量除以执行的指令总数。EPI 是通过在一个任务花费的能量除以执行的指令总数来计算的。它对性能的度量与 CPI 非常相似, 但又解决了功耗和能量问题。EPI 节流 (EPI throttling) 是一种技术, 其中每条指令所花费的能量适应于可以并行执行的指令的数目。当并行指令的数量有限时, EPI 尽可能快地执行它们。当并行度很高时, EPI 尽可能地降低执行指令数, 从而保持在功率范围内。

2.6 可靠性

工艺变更的一个负面影响是降低了器件的可靠性, 从而导致计算机系统中的各种故障。故障大致可分为三种类型: 瞬时故障, 间歇性故障, 永久故障。每个故障类别可以体现在多个方面。瞬时故障引起非破坏性的位翻转 (从 1 到 0 或从 0 到 1), 仅仅是错误的数据值, 除了一次性的位翻转以外对器件没有影响。间歇性故障在某种条件下导致时序或逻辑违规, 如高温, 其中的逻辑电路的计算值没有传播到输出锁存器。间歇性故障可能会持续一段时间, 影响电路的行为, 但没有长期的影响。瞬时和间歇性故障在不同的粒度是可恢复的。然而, 永久故障是不可恢复的, 如 stuck-at-1/0 故障, 不论电路输入如何, 输出锁存器被永久地停留在 1 或 0。当检测到一个永久的故障时, 电路是不可用的, 必须进行分离并从操作中移除。

2.6.1 故障和错误

由于设计一个可靠的系统是有代价的, 系统设计人员必须弄清楚故障 (fault) 和错误 (error) 之间的区别。显然, 避免故障是首要的设计目标。然而, 完全的无故障可能是不现实的, 甚至在某些情况下是不必要的。故障是可靠性降低的物理表现, 但它并不一定转化为一个错误。cache 行的单比特翻转是一个故障, 但如果处理器从来没有访问故障位, 它不能转化为一个错误。一个故障可以通过多种检测和校正技术被限制在一定范围内。因此, 可靠性的另目标是防止故障破坏计算。例如, cache 中的比特位翻转可被检测出, 并用单纠错双检错 (SECDED) 码校正。如果比特位翻转在处理器访问该位之前被校正, 则该故障对程序的结果没有影响。在这个例子中, 故障隔离的范围是在 cache 内, 并且只要故障是缓存外部不可见的, 那它就只是一个故障, 并没有转化为一个错误。

如果在同一 cache 行有两位翻转 (两个故障), 则一个 SECDED 代码无法纠正这两个错误, 尽管它检测到该 cache 的数据是错误的。因此, 处理器知道该值不正确。在这种情况下, 故障已越过缓存范围, 并已在可见的范围外, 且故障已经成为一个错误。如果 cache 行有三个故障位, 在 cache 中的 SECDED 代码甚至无法检测到一个错误的存在。因此, cache 行跨越了缓存范围的界限, 故障最终变成了错误。在 SECDED 可以检测到的错误和 SECDED 不能检测到的错误之间, 有着根本的区别。在第一种情况下, 当处理器访问数据时, 它知道它访问了错误的值。当 SECDED 不能检测到错误, 则处理器不知道它访问的数据是错误的。

根据影响, 故障可划归三类错误类型。包含在容错范围内的故障是可以纠正的, 这些故障被称为可恢复的错误 (Recoverable Error, RE), 比如使用误差校正机制控制结果的产生。如果检测到错误, 但不在容错范围之内, 错误变为能被检测到但不可恢复的错误 (DUE)。如果故障不在容错范围, 甚至没有被检测到, 它就变成了无记载数据损坏 (Silent Data Corruption, SDC) 错误。可恢复的错误不会影响系统的行为, 而 SDC 错误是最具破坏性的错误, 因为该系统甚至不知道它们。虽然 SDC 错误极为罕见, 但它可能对用户造成非常严重的后果。

系统可靠性的重要程度是与发生故障的概率 (非常小的) 和一个类型的故障可能对用户

造成的影响相关的。即使故障的概率非常低，但一个错误的成本可能会非常高。考虑到处理器平均每 5 年一次遇到不可恢复的错误（DUE 或 SDC），而且平均每年数百万的处理器都在使用，则每年五分之一的处理器预计会出错。平均每天超过 500 多个不可恢复的故障发生，并且可能造成灾难性的后果，这对于任何一个消费类电子公司都是一个残酷的现实！所以，每 5 年发生一次无法恢复错误的故障率是不可接受的，虽然它对每个用户来说似乎可以接受。

故障和错误之间的区别取决于容错的范围。这里是一个没有正式定义的边界。cache 行故障的发生可能引起 cache 设计师的关注。如果没有任何形式的保护，cache 中的故障将无法被检测到，并且 cache 的 SDC 错误还会传播。cache 设计者决定在容错中添加奇偶校验或 SECDED 校验，或至少减少 cache 范围外的 SDC 故障率。逻辑设计人员可能担心流水线阶段内时序冲突。为了防止时序冲突，设计者必须加快那些电路中延迟接近于时钟周期的脆弱路径。在这种情况下，故障的范围在一个流水线级内。同样，通过检测错误的计算、在 CMP 中的不同核心恢复和重新执行代码等，一个操作系统可以容忍片上多处理器（CMP）的错误。在这种情况下，容错范围是整个处理器，只有当在 CMP 中的所有核心产生错误时，应用程序才可能会遇到一个错误。

图 2-8 展示出了 cache 范围之内的三个类别的故障。图 2-8a 所示 cache 行发生的有 SECDED 保护的单比特位翻转。因此，当该位在 cache 的范围之外被处理器读取时，错误是可恢复的。两比特位翻转导致 DUE，如图 2-8b 所示。这是一个被检测到的不在容错范围内的错误。最后是三比特位翻转故障引起的 SDC 错误，如图 2-8c 所示。

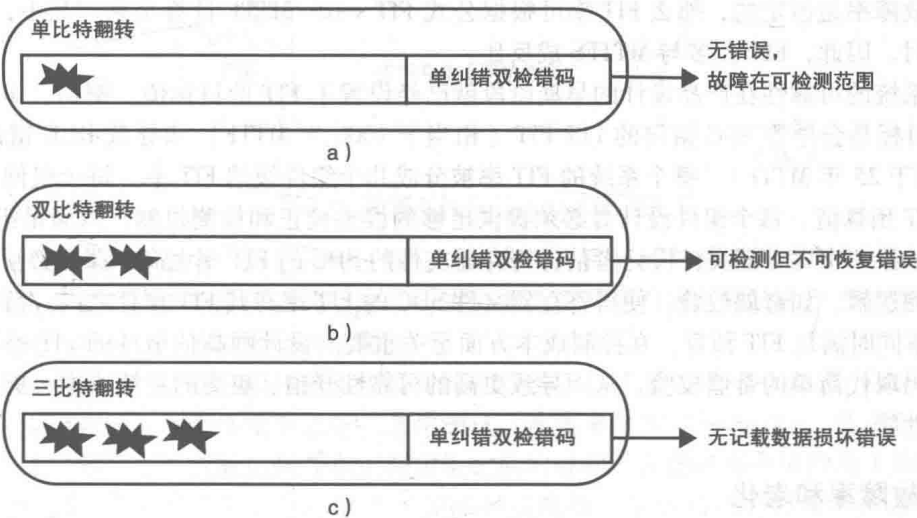


图 2-8 在一个 cache 行的容错

2.6.2 可靠性指标

可靠性不像功率和性能那样可以通过度量诸如瓦特和每个指令周期来被精确量化，它只能通过统计方法测定。组件的可靠度 $R(t)$ 是时间的函数，是由 $R(t) = N_s(t)/N$ 算得的预期估计值，其中， $N_s(t)$ 是到时刻 t 幸存的组件数量， N 是组件取样的总数目。可以从这个简单的公式中看出， $R(t)$ 依赖于时间，但更重要的是它依赖于样本，这意味着可靠性测量是统计性的。

工业上用于测量元件可靠性的一个通用指标是平均无故障时间（Mean Time To Failure, MTTF），即一个幸存的系统不发生故障的平均时间。鉴于目前的计算机系统一般可以可靠地工作数年，MTTF 通常以年来衡量。

为了准确地计算 MTTF, 系统设计者必须测量系统的样本群体出错的时间, 并且一直等待直到系统中所有的样品群体都出错。这两个制约因素构成显著挑战。为了说明这一难度, 让我们考虑三个计算机系统的样本群体。我们假设, 系统设计者观察这个样本 2 年。经过整整一年第一个系统出现故障, 第二个系统在第二年的年底出错, 第三个系统在第二年底仍然运作。如果设计者只考虑为期两年的观察时间内的错误, 这两个系统的 MTTF 为 $(1+2)/2=1.5$ 年。观察样本中两年后仍旧能使用的第三个系统, 设计师不知道它的 MTTF。如果第三个系统在第三年底恰好出错, 则 MTTF 为 $(1+2+3)/3=2$ 年。因此, 设计者必须找到估计在观察期结束时幸存的系统的 MTTF 数目的最佳方式。如果出错率是随时间恒定的, 就可以通过假设幸存系统在未来也将以同样的速度出错来估算 MTTF 的值。

由于电脑系统可能有很长的寿命, 让一个系统设计者为等待确定的 MTTF 观察故障多年是不现实的。为了在更短的时间间隔内观察和测量故障, 系统设计师将系统置于极端条件下进行测试, 比如高温、更高的芯片频率和高电压。根据高强度测试条件, 设计师可以推断出系统操作在正常条件下观察到的 MTTF。

另一种常见的可靠性指标是 FIT (failures-in-time) 率。一个系统的 FIT 率是在系统中 10 亿小时 (超过 100000 年) 观察到的发生故障的平均数目。FIT 是一种度量计算机系统可靠性的比 MTTF 更为方便的指标。计算机系统是组件的集合, 每个组件都有自己的 FIT。整个系统的 FIT 是通过将所有组件的 FIT 率简单地累加在一起计算的。因为 FIT 具有累加性, 才使 FIT 超越 MTTF 成为系统设计上可靠性的首选指标。

如果故障率是恒定的, 那么 FIT 率可根据公式 $FIT = 10^9 / MTTF$ 计算出来, 其中, MTTF 的单位是小时。因此, FIT 大多与 MTTF 成反比。

整个系统的可靠性在产品设计的早期阶段就已经设置了 FIT 的目标值。例如, IBM 未来系统的 FIT 目标是会导致 SDC 错误的 114 FIT (相当于 1000 年 MTTF) 和导致 DUE 错误的 4566 FIT (相当于 25 年 MTTF)。整个系统的 FIT 率被分成几个组件级的 FIT 率。每个组件都被分配了一个 FIT 预算值。每个组件设计者必须提供足够的误差校正和检测机制, 以满足组件的 FIT 预算值。如果一个寄存器文件设计者估计寄存器文件的 SDC 的 FIT 率太高, 那么必须增加额外的错误检测逻辑, 如奇偶校验, 使得寄存器文件 SDC 的 FIT 率在其 FIT 预算之内。精确测量组件的 FIT 率同时满足 FIT 预算, 在控制成本方面至关重要。设计师高估组件的 FIT 率可能会增加纠错代码取代简单的奇偶校验, 从而导致更高的可靠性开销、更高的系统成本、更高的功耗和较低的性能。

2.6.3 故障率和老化

部件的可靠性 $R(t)$ 是一个组件生存到时间 t 的概率。它是时间 t 的函数, 用 $R(t) = N_s(t)/N$ 计算, 其中, $N_s(t)$ 是存活到时间 t 的组件的数量, N 是组件的总数; $N_f(t)$ 是未能生存到时间 t 的组件的数目, 等于 $N - N_s(t)$, 一个组件到时间 t 出现故障的概率为 $Q(f) = N_f(t)/N$, 所以 $Q(t) = 1 - R(t)$ 。

为了测量真实的可靠性, 必须有一个无限数量的部件, 即 $N \rightarrow \infty$ 。然而, 在现实中是不可能测量无限多的组件的可靠性的, 所以 $R(t)$ 通常从大小为 N 的一个相当大的抽样中获得。

请注意, $Q(t)$ 是一个概率分布函数, $f(t)$ 是在时刻 t 出错的概率密度函数, 下面给出 $Q(t)$ 的导数:

$$f(t) = \frac{d}{dt}Q(t) = -\frac{d}{dt}R(t) = -\frac{1}{N} \times \frac{d}{dt}N_s(t) \quad (2.16)$$

在时间 t 时的预期故障的总数是 $f(t) \times N$ 。这种故障密度概率是无条件的。这表明 N 个初

始组件中故障的瞬时数量随着时间如何变化。一个组件只有当它存活到了时间 t 才可以在时间 t 出错。在时间 t 只有 $N_s(t) = R(t) \times N$ 个组件仍然存在。所以幸存组件在时间 t 故障的概率是通过将 $f(t) \times N$ 除以 $N_s(t)$ 得到。 $h(t)$ 被称为故障率 (failure rate) 或风险函数 (hazard function), 其公式如下:

$$h(t) = \frac{f(t)}{R(t)} = \frac{1}{R(t)} \times \frac{d}{dt} N_s(t) = -\frac{1}{R(t)} \times \frac{d}{dt} R(t) \tag{2.17}$$

这个风险函数是非负的。

通常情况下, 故障率与时间的函数遵循“浴盆”曲线, 如图 2-9 所示。早期故障率高是因为有缺陷的组件或者是组件的工作量很小, 但它随后迅速下降。如果组件在这个阶段存活, 那么就进入了真正有用的生命期, 故障率恒定且非常低。在这段时间内, 可以假设在恒定故障率条件下故障发生是随机的。换句话说, 故障的偶然发生服从泊松过程。当组件在其真正有用的生命期内, 组件的 FIT 率是累加的, 因为多个泊松过程的合并也是一个泊松过程, 其速率等于所有进程的速率和。



图 2-9 随着时间而变化的组件故障率

在一个组件的生命中, 由于种种过程它会被磨损, 我们将在本章后面介绍。由于磨损, 组件达到其生命周期的第三阶段, 这个阶段的故障率会由于陈旧的电路故障激增。在这一点上该组件已经到达它设计的使用寿命的结束。

为了避免发布仍处于早期故障率较高阶段的产品, 一个被称为老化测试 (burn-in testing) 的过程被应用到每一个组件发布之前。老化测试, 或简单老化 (burn-in), 是一个通过大量使用一个组件以测试它是否能达到其真正有用生命期的过程。在此过程中坏的或工作量少的部件都将被丢弃。老化测试是在压力的条件 (如高温或高压) 下进行的, 以检测出那些在极端的环境条件下没有弹性的组件。老化测试可以避免早期故障阶段。

例 2.1 假设一个部件的故障率是常数并且等于 λ , $\lambda = 0.001$ 。一批 100 万组件样本当中 5 年内的出错预期是多少?

从式 (2.17), 有

$$h(t) = -\frac{dR(t)}{dt} \times \frac{1}{R(t)} = \lambda \tag{2.18}$$

$$\int_0^T \lambda dt = \int_0^T -\frac{dR(t)}{dt} \times \frac{1}{R(t)} dt \tag{2.19}$$

$$\lambda T = -\log R(T) \tag{2.20}$$

或

$$R(T) = e^{-\lambda T} \tag{2.21}$$

五年之后给定的组件出现故障的概率为 $Q(5) = 1 - R(5) = 1 - e^{-0.005} = 0.004988$ 。因此预计在 5 年内这批 100 万组件中出错了的组件的数量是 4988 个。请注意, $h(t)$ 很容易被等同于 $f(t)$, 从而使预期故障数是 5000, 不是 4988, 12 太多了! 这显示出了两个概率之间的差别。◀

2.6.4 瞬时故障

随着每一代生产工艺的发展, 存储在每个存储单元 (DRAM、SRAM 或寄存器) 的电荷量因为电源电压 V_{dd} 和其电容的变化而减小。由于这种趋势, 存储器单元变得越来越易受高能粒子的侵害而导致瞬时故障, 有时也被称为软错误。瞬时故障不同于由损耗而导致的典型的故障, 它是半导体电路中特有的。它们不是组件生命周期的一部分, 如图 2-9 所示。瞬时故障是一次性事件, 并且不留任何可以纠正错误的痕迹。

瞬时故障是由高能粒子撞击造成的。这类攻击通常被称为单粒子翻转 (Single Event Upset, SEU)。SEU 有两种常见的原因: 第一个原因是从芯片的封装内产生 α 粒子撞击, 这些颗粒是由放射性原子核释放出来的, 例如由存在于芯片封装和焊锡球的杂质所释放出的镭; SEU 的第二个原因是中子撞击, 中子撞击主要是由宇宙射线的外层空间的活动引起的, 这些宇宙射线携带高浓度的质子, 而当质子与地球大气相互作用, 它们产生大量的中子散射。中子通量指标测量了每秒穿过 1cm^2 的中子数量, 中子通量依赖于高度和地球磁场强度, 是随位置的变化而变化的。例如, 在海拔为 3 万英尺的中子通量大约是在海平面的中子通量的 220 倍, 在海平面上操作计算机系统时中子打击比在空中操作更少。

图 2-10 显示了一个单粒子翻转的简单例子。高能量中子撞击 nMOS 晶体管。粒子穿过 p 衬底的硅。随着粒子穿过衬底, 它通过与硅原子核相互作用产生了电子空穴对。通过这些电子空穴对产生的电荷在扩散区生成了驱动电子或空穴的电场, 从而产生感应电流。如果在 SRAM 单元中的电荷小于累积电荷, 由感应电流累积的电荷会翻转 SRAM 单元的值。存储在内存中的电荷越弱, 粒子打击越容易翻转存储的值。翻转 SRAM 单元的内容所需的电荷量称为临界电荷 (Q_{critical}):

$$Q_{\text{critical}} \propto C_{\text{node}} \times V_{\text{dd}} \quad (2.22)$$

其中 C_{node} 是节点电容, V_{dd} 是电源电压。在每一代工艺发展中, C_{node} 和 V_{dd} 都会下降 (见表 2-1 和表 2-2), 因此 Q_{critical} 以迅猛的速度减少。

瞬时故障的另一个来源是电气噪声。电气噪声可以由在总线的信号或电源分布网之间的串扰造成。当信号通过控制逻辑和处理器的功能单元中的随机逻辑传播时, 它们也可能由于彼此之间的串扰被损坏。下面我们重点介绍粒子撞击造成的 SEU。

单粒子翻转率 (SER) 可以基于众所周知的 Hazucha 和 Svenson 模型来计算, 公式如下:

$$\text{SER} = k \times \text{flux} \times \text{bitarea} \times e^{-Q_{\text{critical}}/Q_{\text{collect}}} \quad (2.23)$$

其中, k 为常数; flux 是中子通量; bitarea 是对软错误攻击敏感的区域面积; Q_{critical} 是临界电荷; Q_{collect} 是 SRAM 单元的电荷收集效率。要注意的是 bitarea 和 Q_{collect} 仅依赖于工艺技术。特别是, 这些参数独立于操作环境。另一方面, Q_{critical} 依赖于工作电压和节点电容。例如, cache 中的动态变压变频 (DVFS) 影响 Q_{critical} 的值。

给定 SER, 在 T_c 时间内一个时钟周期发生单粒子翻转的概率 p_{SE} , 是由一个时钟周期内发生翻转的次数决定的。这个数量符合泊松分布, 并且 λ 与由式 (2.23) 得到的 SER 相等。一

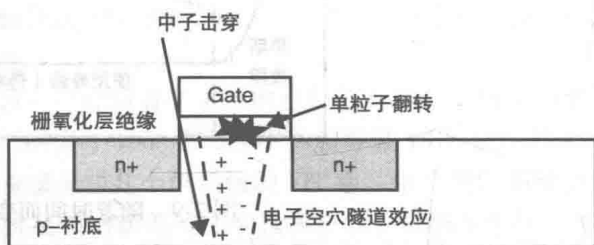


图 2-10 在晶体管内的单粒子翻转

个周期内奇数次位翻转则导致故障，否则就是正确的。这样我们就可以精确地计算概率 p_{SE} ，如下：

$$p_{SE} = \sum_{\text{odd } k} \frac{e^{-\lambda T_c} \cdot (\lambda T_c)^k}{k!} \quad (2.24)$$

将来，一些反作用力将影响 SEU。无论是 T_c 或 bitarea 在每一代工艺进步中都减少，使得位翻转的概率降低。然而， $Q_{critical}$ 也在每一代工艺进步中减小。尽管 T_c 和 bitarea 的趋势是有利的， λ 与 $1/Q_{critical}$ 的指数关系占主导地位，再加上越来越多的存储器位都挤在一个芯片上，这样会导致在未来瞬时故障率显著地增加。

cache 的瞬时故障

由于大多数现代微处理器的芯片面积的约 50% 通常被 cache 占用，设计者相当重视保护缓存避免瞬时故障。如果在 cache 行，产生瞬时故障的故障位由处理器读取（无论是作为一个取指令或是由负载返回的值），则故障变为错误。

保护 cache 避免瞬时故障常见的方法有错误检测和纠错码。将单比特的奇偶校验添加到每个 cache 行可以检测一比特位翻转，从而把潜在的 SDC 错误转化为一个 DUE。一个 SECDED 代码可以纠正单比特错误和检测双比特错误。在这种情况下，单比特瞬时故障甚至不能作为故障离开 cache 域。这是计算机设计师的选择保护级别的责任。

这个选择取决于以下因素：

(1) 了解问题的重要性显然是首先要考虑的。因此，瞬时故障的概率 p_{SE} 是第一考虑。据估计，3GHz 的条件下，65nm 工艺的瞬时故障的概率大约是 10^{-25} / (bit 周期)。不发生瞬时故障的概率为 $1 - p_{SE}$ 。假设缓存设计师被给予了 FIT 预算，设计师必须将 p_{SE} 换算成 FIT 率。给定的 p_{SE} ，一简单的来计算含有 N 比特位 cache 的 FIT 率的方法是，计算在高速缓冲中存储器在一个周期内至少有一个故障的概率 $1 - (1 - p_{SE})^N$ ，也就是说对于 1 兆位的 cache 在一个周期内至少有一个故障的概率为 $1 - (1 - p_{SE})^{1024 \times 1024}$ ，换算后为 1150 FIT。然而，这种 FIT 率被严重高估，因为它假定 cache 行的任何一位故障都会导致计算错误。实际上，一个损坏的 cache 行可能是“死的”（也就是说，在它被删除之前它永远不会被再次访问），或者故障位无法被处理器读取和使用，也不会使计算出错。基本 FIT 率被称为本征（intrinsic）FIT 率。本征 FIT 率是用来粗略决定设计人员可以选择的保护类型的第一阶度量。

(2) 保护的面积开销是第二考虑因素。奇偶校验的面积开销是每个受保护的数据块 1 位。假设一个缓存每个 64 字节数据块使用 1 位校验位，面积开销为 4/512（小于 0.2%）。SECDED 开销随被保护数据的大小改变。对于一个 64 字节的 cache 行，SECDED 开销为 11 位，2% 的比特开销。SECDED 码保护每一个 32 位字需要 7 位或者超过 20% 的比特开销。

(3) 除了位的开销，保护也可能会增加 cache 行的访问等待时间，因为错误检测或校正必须在处理器使用的数据之前进行。就像位开销，SECDED 检查需要的时间随每个码保护的字节数改变。例如，检查 SECDED 代码保护 64 字节行可能会在 3GHz 处理器中增加三个周期的 cache 访问延迟。

体系结构的脆弱性因素 (AVF)

本征 FIT 率高估了一些硬件的脆弱性，从而导致设计师过度提供硬件保护。例如，cache 中不是每个 SRAM 单元都会转换成一个计算错误。一个 SEU 的影响可能被各种各样的原因掩饰，例如，损坏的 cache 行是无效的，在读取之前它已经被覆盖了，它是空的或者在 cache 行中的块没有被再次引用。AVF 分析技术将这些因素考虑进去，改善了本征 FIT 率的严重高估的问题。该 AVF 表征了在存储单元中发生一个用户可见的比特翻转错误的概率。

AVF 分析的基本前提是，只有当它变得对用户可见时，位翻转才成为一个错误。由于微架

构状态是用户不可见的，在微架构的任何错误都可以忽略不计，除非它传播到体系结构状态。例如，在分支预测表中，硬件的粒子撞击不影响执行的正确性，但它会导致不好的分支预测。

在程序执行期间，根据其是否影响了执行，每个比特被分类为结构上相关的或不相关的。如果一个位影响到执行的结果，则被认为是 ACE（即要求体系结构正确地执行），否则它是 un-ACE。一个基本的存储单元（一位存储部件，如一个触发器、一个 SRAM 单元或者 DRAM 单元）的 AVF 是一个包含 ACE 位的时间的一部分。当存储单元的碎片时间里包含一个 un-ACE 位时，单元上的任何瞬时性故障在体系结构级别是无害的。多比特的硬件结构的 AVF 是所有的基本存储单元的 AVF 的平均值，例如 cache 或者程序计数器。计算结构的有效 FIT 率时，用它内部的 FIT 率简单地与其 AVF 相乘（或降级）。从本质上说，在一个位的计算完整性影响了 AVF 分析。空间脆弱性（与位架构相关的）和时间脆弱性（位曝光会持续多久）有助于改变内在 FIT 率大小，使 FIT 率与硬件的正确性相关更强。

L1 cache 位单元的 AVF 分析

为了进一步说明 AVF 的计算，考虑在 L1 cache 行影响位单元的事件的顺序，如图 2-11 所示。存储器块在时间 T_0 被加载到 cache 行。存储在 cache 行中的位由处理器在周期时间 T_2 更新。在时间 T_5 处理器读取位（取指令或数据），这是指令提交执行的一部分。在时间 T_7 位于该行的块被删除。位单元在 T_1 、 T_3 和 T_6 被翻转三次。在时间 T_1 翻转不影响执行，因为在时间 T_2 有故障的位不能被处理器读出并覆盖。在时间 T_6 的翻转不影响执行，因为位在该块被替换之前不能被处理器访问。然而，在时间 T_3 的位翻转导致一个错误，因为错误的值破坏了时间 T_5 的执行。

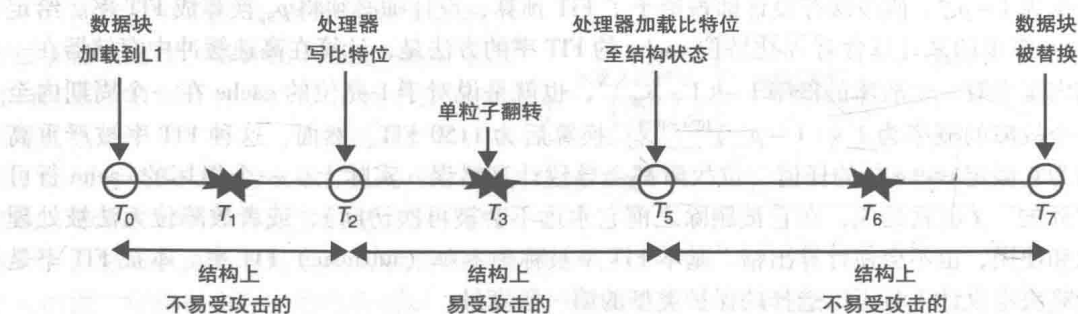


图 2-11 L1 cache 行中的位单元事件的时间轴

本征 FIT 率假设该位在周期 $T_7 - T_0$ 的执行窗口是脆弱的。而 AVF 分析认为该比特只在周期 $T_5 - T_2$ 的窗口内是脆弱的。因此，位单元的 AVF 用 $(T_5 - T_2) / (T_7 - T_0)$ 计算。然后，位单元的本征 FIT 通过位固有的 FIT 率与 AVF 相乘得到新的 AVF-derived。

2.6.5 间歇性故障

故障的第二类包括芯片使用期间因设备的老化和环境条件下产生的压力而造成的间歇性故障。设备老化也被称为耗损。随着时间的推移晶体管老化，它们的电气特性慢慢降低，如 $I_{DS,sat}$ ，耗损速度随时间变缓慢，在某些情况下甚至可以逆转。

热应激条件或极端电压会加速装置劣化，并造成芯片的时序故障，甚至发生随机比特翻转。这些故障之所以被称为间歇的（intermittent），是因为它们可能会持续到芯片温度下降或电压上升。间歇性故障可以持续一段时间，然后消失。因此间歇性故障的电路应暂时停用，直到造成故障条件被移除。

间歇性故障依赖于芯片已经运作的时间和其他数个物理现象。在此我们将介绍一下导致间

歇性故障最常见的物理现象。本节的目标是在很高层次上从体系结构的角度概述，而不是描述这些故障背后的物理学。

电迁移

众所周知，电迁移（Electro Migration，EM）是导致处理器的线中间歇性故障来源。线的尺寸随每代工艺减小，如导线宽度从 65nm 工艺技术的 120nm 降低到 35nm 工艺技术的 60nm。随着线宽度减小，导线内的电流密度增大。如果把电流密度放在现代微处理器内部，则现在一个典型的服务器处理器工作电压为 1V 时消耗超过 100W 的功率，导致处理器的电流达到 100A ($P = VI$)。给定极窄的导线尺寸，在导线中的电流密度可达到每平方厘米百万安培。在这种极端的电流密度的条件下，电线中的金属原子与电子之间的碰撞加快，并且向电子流的方向移动。如果电流是单向的，那么，由于电子的持续推动，一些金属原子永久地迁移到电线的另一端。金属原子的单向电流流动运动被称为电迁移。

EM 的效果是，该金属在导线的一端原子枯竭，而这些原子在另一端聚集。枯竭导致一个空白，并积累起来产生堆集（hillock deposit）。随着空白区域尺寸的增加，它最终把导线切割为两部分，由此引起了开路故障。同样，随着堆集大小的增加，失效的堆最终可能会触及相邻的导线，从而导致短路故障。在一个永久性的开路故障发生之前，流过耗尽区的电流遇到阻值较高的线电阻会放热，提高温度并减慢电路速度。随着电阻的增加，若电压是固定的，电流会减小，从而可能导致间歇性故障。例如，当导线向多个门输送，并且电流逐渐下降到不能够驱动所有的门的一个点，那么通过线接收数据的一个电路可能会失效。

例 2.2 在金属层之间的孔的电迁移 虽然任何电线都易受 EM 影响，但在一个处理器通信结构中连接不同的金属层的过孔通常被认为是最容易受到 EM 影响。处理器内部子部件通过排列在几个金属层（一般 7~10 的金属层）之间的导线进行通信。通常，层间的通信是通过有大量单向电流流过的金属过孔来实现的，如图 2-12 所示。

孔 1 内电流的流动在金属层 1（M1）和金属层 2（M2）之间是单向的。由于电子从 M1 通过孔 1 流向 M2，通过减少孔 1 的有效宽度可将空隙创建出来。另一方面，当电子从 M2 流向 M1，它们推动金属原子在孔 2 创造了堆。

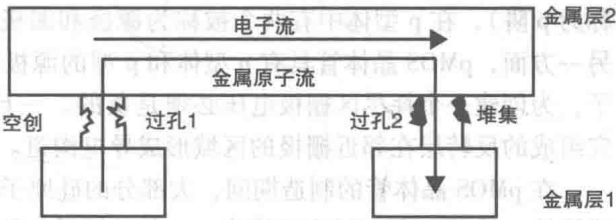


图 2-12 在孔中电迁移的效果

例 2.3 在 SRAM 单元的电迁移 图 2-13 显示了存在于处理器设计的单向电流的另一个例子。图中表明一个 6T 的 SRAM 单元适用于处理器的几乎所有的存储结构，例如寄存器堆，重排序缓冲区，以及加载/存储队列。这个 6T 单元是由 4 个交叉耦合的反相器和两个作为传输的晶体管组成，在图中标注为 P1 和 P2。传输晶体管被连接到一对位线（BL 和 \overline{BL} ）。位线在顶部连接了预充电逻辑，在底部连接了一个读出放大器。在读取操作期间位线首先被预充电到 V_{dd} 。如果内存单元存储了 0，那么晶体管 P1 允许电流通过，而 P2 不允许。如果内存单元存储为 1，那么晶体管 P2 允许电流通过，而 P1 阻止。在两个传输晶体管中不同电流流动引起位线之间的电压差。通过读出放大器检测两个位线之间的电压差来检测 0 或 1。

在这里要注意的是，无关紧要的值被读出，电流总是在连接位线到传输晶体管的电线中沿相同方向流动，即从位线流向存储器单元。由于性能的原因，大多数处理器使用不同的导通晶体管用于读取和写入存储单元。例如，单一的问题处理器可能需要两个读端口和一个写端口访问寄存器堆。两个读端口用两对导通晶体管实现，这种晶体管与用于写入寄存器文件的导通晶

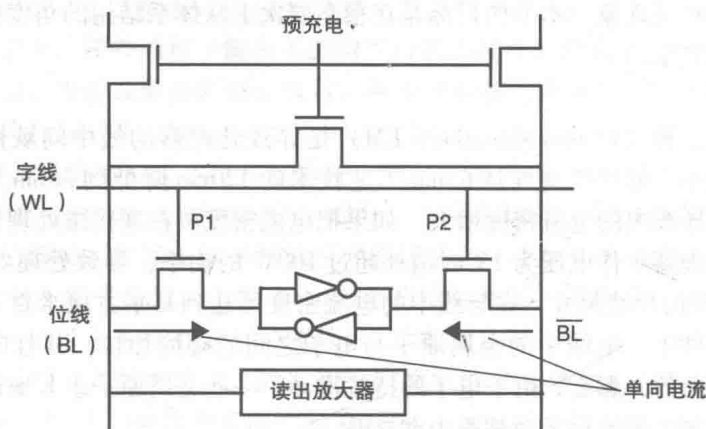


图 2-13 在 SRAM 单元电迁移效果

晶体管对是相互独立的。因此，每次有端口上的读操作时，具有两个读端口的四个传输晶体管在单向电流的压力下是有效的。随着时间的推移，该连接位线到读出端口的旁路晶体管的导线受到电迁移的影响，导致间歇性的电路故障最终演变为硬件故障。

如果电流的流动可以改变，则电迁移效应可以被恢复。然而在许多情况下改变电流流动的方向是不切实际的。在例 2.2 中，电流不能在读取端口改变。然而，如果相同的导通晶体管是读写共享的，那么写 0 或 1 可以改变两个传输晶体管中一个的电流流动的方向。共享读写端口可以用于提高可靠性。

负偏压温度不稳定性 (NBTI)

NBTI 主要关注 pMOS 器件的可靠性。参看图 2-2 中的说明，nMOS 晶体管使用 p 型体（也称为 p 阱），在 p 型体中有两个被称为源极和漏极的重掺杂的 n 型区域被扩散到 p 型衬底中。另一方面，pMOS 晶体管具有 n 型体和 p 型的源极和漏极。在 pMOS 晶体管中多数载流子是电子，为创建一个耗尽区栅极电压必须是负的。一旦栅极电压小于（负的）阈值电压时，正空穴组成的反转层在邻近栅极的区域形成导电沟道。

在 pMOS 晶体管的制造期间，大部分的硅原子结合氧原子在栅极与 n 型体之间形成了 SiO_2 绝缘体。然而，由于制造不精确，一些硅原子与氢原子在硅衬底和栅氧化层的交界处结合。当 PMOS 器件具有负的栅极偏压，即 $V_G = 0$ ， $V_S = V_D = V_{dd}$ 时，若器件的工作温度较高，那么氢原子会与硅分离。由于栅极带有负电荷性，上述的氢原子向栅极漂移，悬空的硅原子留在在硅衬底和栅氧化物之间。悬空硅原子带着由氢原子留下的空穴漂向栅极。在栅氧化物和硅衬底的边界处，空穴聚集产生了空穴的导电沟道。当在栅极加负偏置电压时，即 $V_G = V_{dd}$ 且 $V_S = V_{dd}$ ，氢原子又被吸引到悬空硅原子的周围，导电通道消失，从而从 NBTI 的影响中恢复。如果门是正偏压，即 $V_G = V_{dd}$ ，且 $V_S = 0$ ，会使氢原子流向硅的过程加速。

图 2-14a 表示 NBTI 的应激阶段。当 $V_G = 0$ 时，氢原子漂向栅极，在栅氧化层和衬底的交界处形成空穴的反转层。恢复阶段如图 2-14b 所示。当 $V_G = V_{dd}$ 时，大部分的氢原子朝硅回迁。然而，恢复并不完整，因为一些氢原子永久地留在了栅极，从而使得空穴在衬底缓慢地积累。

恢复阶段后留下的空穴吸引着在 n 阱栅极氧化物的电子。因此，下一次 pMOS 晶体管导通时，为形成在 n 阱中排斥电子的传导通道高电压是必需的。该电压的增加实质上提高了阈值电压。因此，随着时间的推移，由于负偏压的反复施加，阈值电压逐步降低，并且该晶体管的开关速度也降低。在所有处理器中的不同的微架构结构中，cache 是最容易受到 NBTI 影响而减速的。SRAM 缓存具有密集的 pMOS 器件（每比特两个 pMOS 器件），以及许多储存为零的缓存

位，因为大多数数据项的高阶位为零。

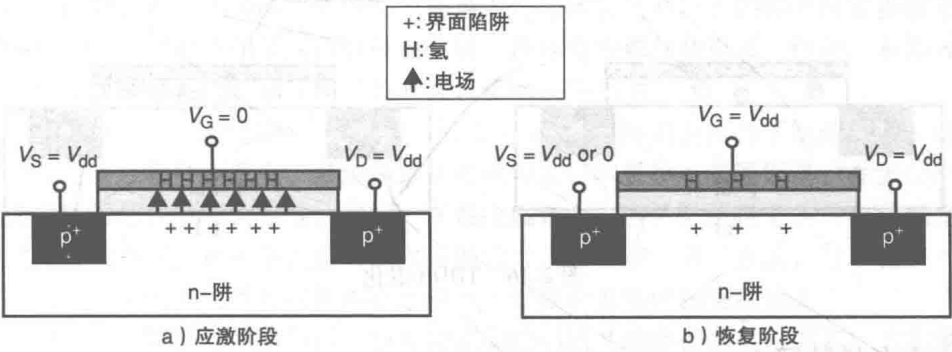


图 2-14 pMOS 的 NBTI 效应

图 2-15 显示了 NBTI 应激因素被移除时，部分阈值电压的恢复过程。从时间 T_0 到 T_1 ，pMOS 器件在高温下经历了栅极到源极的负偏压。因此，阈值电压开始增加。在 T_1 到 T_2 之间，负偏压被移走。在此期间，该 pMOS 器件迅速开始恢复，并且阈值电压开始降低，但它并不能完全恢复基本的阈值电压。在 T_2 时刻，当负偏压被重施加，阈值电压再次增加。如图 2-15 所示，在每个应激和恢复周期，阈值电压随着时间的推移逐渐增加。

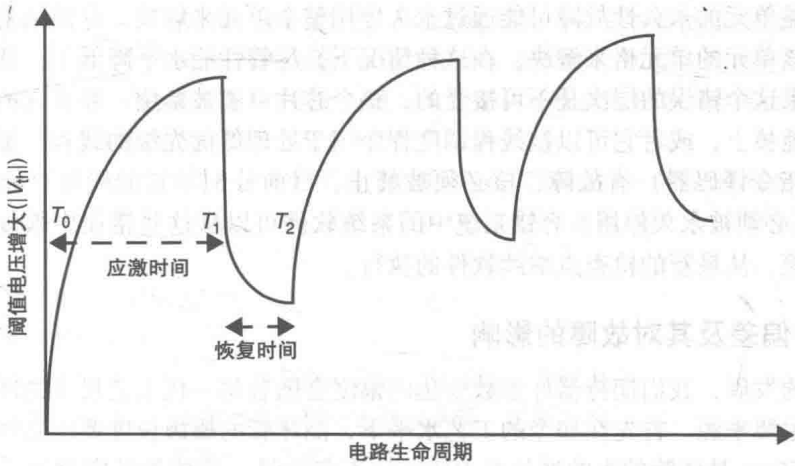


图 2-15 NBTI 的部分恢复

与时间相关的介质击穿 (TDDB)

TDDB 是由栅极电介质材料在较长的时间内损耗造成的击穿，它在栅极氧化层中产生了导电通路。图 2-16a 和图 2-16b 说明了 TDDB 造成的渐进性退化。晶体管的栅极氧化层被认为是绝缘体，这意味在有栅极氧化物内没有电荷载体。由于多种原因，如老化和制造不精确，陷阱在栅氧化层缓慢积累。这些陷阱在图中被表示为在栅极氧化物的孔。随着每一代加工工艺中晶体管的尺寸缩小和电源电压的降低，为了在晶体管内部形成具有更低电源电压的导电沟道，栅极氧化层的厚度也减小到的只有几个原子厚。因为栅极氧化层的厚度减小，在栅极与衬底之间形成导电路径只需要连接几个孔。电流流过栅极与衬底之间的这些导电路径，导致介电层的击穿。栅极泄漏电流使源极和漏极之间的主要驱动电流减小了。驱动强度的减少会减慢装置，最终导致时序违规。与 NBTI 和电迁移不同，TDDB 退化是不能自我恢复的，并且最终导致永久性设备故障。

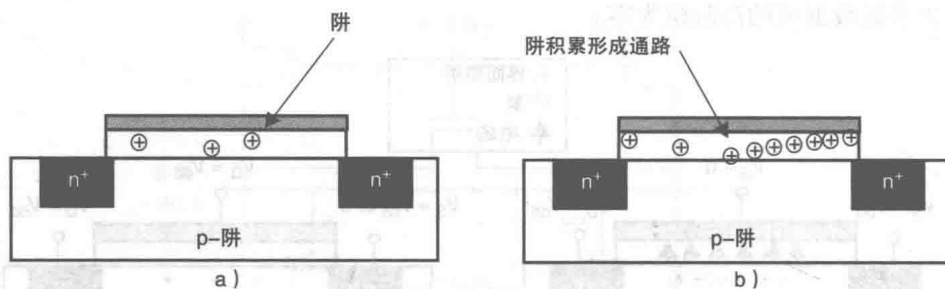


图 2-16 TDDDB 退化

2.6.6 永久性故障

故障的最后一类是永久性故障，诸如 stuck-at-1 和 stuck-at-0 故障。永久性故障的原因可能是芯片生产制造不精确。制造缺陷在老化过程中得以确定。永久性故障也可能是一个间歇性故障持续的后果。例如，如果单向电流持续通过一种金属通道，电迁移引起的金属原子迁移最终造成裸线，导致永久性故障。间歇性故障向永久性故障的演变在老化期间很难被检测到。很多我们所讨论的间歇性故障发生在应激周期后，即使加强老化过程也不能轻易复现。

当检测到永久性故障，逻辑块必须被隔离并停用。该块的关键性决定了采取何种措施。一个核的冗余功能单元的永久性故障可能通过永久停用整个单元来解决。存储结构的永久性故障可以通过禁用该单元的单元格来解决。在这种情况下，尽管性能水平降低了，但处理器将保持正常运行。如果这个错误的层次是不可接受的，整个芯片可能被禁用，并且它的线程迁移到相同处理器的其他核上，或者它可以被线程调度程序用于处理低优先级的线程。最后，如果非冗余核单元（如指令译码器）有故障，核必须被禁止，目前分配给它的线程可能会停止（致命错误），整个核必须被永久停用。容错系统中的系统软件可以从这些错误中恢复，通过定期地检查线程和回滚，从最新的检查点继续软件的执行。

2.6.7 工艺偏差及其对故障的影响

随着技术的发展，我们期待器件参数变化的幅度会随着每一代工艺尺寸的减小而增大。这些偏差有两个主要来源。首先在如今的工艺水平下，晶体管的栅极长度要比光蚀刻装置的波长短。用于蚀刻 65nm 晶体管的光的波长是 193nm。不幸的是，晶体管的宽度和光的波长之间的间隙预计在每一代生产工艺都会增长。这个间隙会导致 within-die 和 die-to-die 的波动。第二，由于特征尺寸的减小，在晶体管沟道中的掺杂原子的数量呈指数降低。例如，在一个 65nm 的晶体管沟道中，掺杂剂原子数只有 100 左右，即使掺杂剂的原子数产生很小的变化也可能显著改变晶体管的电特性。

器件偏差可大致归类为 die-to-die 偏差，随机的（不相关）within-die 偏差，以及系统的（相关的）within-die 偏差。由于 die-to-die 偏差，相同的晶圆制造的芯片彼此有不同的行为。每个芯片可能会有不同量的漏电流，不同的阈值电压，或不同的时间延迟。这种 die-to-die 偏差甚至存在于当前的工艺技术中。业界解决这些 die-to-die 偏差的方案是分级（binning）。分级是将芯片基于它们的表现（主要是根据工作频率）分到不同的性能箱中，并且根据它们的性能箱确定芯片价格的过程。由于在一个箱中的芯片并不一定具有相同的性能，最低的性能芯片实际上代表了整个箱中芯片的性能以及整体收入。因为 die-to-die 偏差不断放大，更多的芯片被放置在更低性能箱中，这会显著影响收入。

die-to-die 偏差是通过分级的方法解决的，然而 within-die 偏差更难解决，这是因为在一个

芯片内, 晶体管的 within-die 波动在芯片的不同部分也许有不同的行为。CAD 工具和内置自测试机制在很大程度上依赖于晶体管的特性, 并且无论晶体管在芯片上的哪个位置都假定给定的晶体管有同样的行为。因为如果这种假设被打破, 设计会变得复杂得多。例如, 不能再通过简单地对每一个阶段中的关键路径的门计数来假设流水是平衡的。

within-die 偏差可以是随机的或相关的。相关 within-die 偏差也称为系统偏差, 在经历了相似的老化过程的临近器件中发生。系统偏差比随机偏差影响要小。其原因是, 在关键路径的设备为避免连接延迟是在空间共存的。因此, 在系统偏差中, 当关键路径上的一个单元受到负面影响, 则所有其他关键路径的单元有可能以类似的方式被影响。另一方面, 与不相关的随机偏差相比, 关键路径中的不同单元可能潜在地具有可以彼此抵消的相反的偏差。

工艺偏差的真正影响是前面所讨论的各种来源的间歇性故障的恶化。例如, 工艺偏差可能导致制造阶段在导线中的金属原子的数目不平衡。当导线是由一个单向电流控制, 就加速了空洞和堆的形成。同样工艺偏差会引起氧化层厚度的微小差别, 若一个特定的设备的氧化物厚度比标称厚度更薄, 该设备比平常更快遭受 TDDB 效应。由工艺偏差引起的间歇性故障会更快引起永久性故障。

习题

2.1 通常用来衡量一个 VLSI 的设计质量的指标一直是面积 \times 延迟 (AD)。然而, 功耗和能耗在设计上导致了新的约束。这种约束体现在新的指标上, 如功耗 (P) (对于热和包装的问题), 能耗 (PD) (对于电池供电的嵌入式系统), 能耗延时 (PD^2) (对于工作站), 以及能耗延时的二次方 (PD^3) (对于高性能系统)。我们基于这 4 个指标对这几个机器的架构进行比较, 参考由式 (2.12) 和式 (2.14) 给出的动态和静态功耗的公式进行解答。

基准机器是一个单周期处理器, 这个处理器会以频率 f 每周期内执行一条完整的指令。

(a) 参考上述基准机器对下列设计 (仅考虑动态功耗) 用 4 个度量 (P , E , PD^2 , PD^3) 估计性能比率:

- 五级流水线主频为 $5f$ 。
- 五级流水线主频为 f 。
- 五路多处理器, 其中每个处理器都是单周期 CPU, 主频为 f 。
- 五路多处理器, 其中每个处理器是五级流水线, 主频为 $5f$ 。

(b) 重新计算 (a), 仅考虑静态 (泄露) 功耗。

(c) 针对不同应用总结并提出设计建议。

2.2 表 2-1 表明了该工艺变更导致电源电压降低 $1/S$ 。不过, 也有明显的迹象表明, 将来进一步的电压缩放就会跟不上这个步伐。

(a) 假设将来电源电压不再降低, 保持不变。根据这一假设, 生成一个具有新的器件和特色的表 2-2。

(b) 请评论在这一新的假设条件下, 哪个设备/线材的特性将进一步恶化, 哪个的特性变得更好。

(c) 假设将来电压和阈值电压都保持不变。根据这一假设, 依据新的设备/线材的特性生成一个新的表 2-2。

(d) 请评论在这一新的假设条件下, 哪个设备/线材的特性将进一步恶化, 哪个的特性变得更好。

2.3 假设在硬件功能块所做的工作相当于 64F04 延迟。为了降低动态功耗, 设计师有一个选择——要么使用流水线, 要么使用并行处理功能块。

(a) 设计人员可以将 64F04 延迟功能块设计为四级流水线。假设供给流水线级的数量呈线性降低。流水线设计的总动态功耗与没有流水线的基准之间有多少差别?

(b) 流水线不是免费的; 它有一个锁存器的延迟开销。假设电源电压与流水线级的数目线性减小,

但在每个流水线阶段有两个 F04 延迟相当于在两个流水线阶段之间浪费了锁存器。如果题中基准设计被分成 8 个阶段, 功耗 (与基线相比) 是多少? 如果设计被分成 16 个阶段, 功耗是多少? 在流水线深度为多少时增加新的流水线级会使增量效益变成负数?

- (c) 设计者选择使用并行处理来代替流水线以降低功耗。如果 64F04 的设计是重复 4 次, 需要多大功耗?
- (b) 并行处理并不总是容易做到。考虑以下情况, 每 N 个并行操作后, 系统需要顺序地执行接下来的 M 个操作。在顺序操作中只有一个被复制的单元可以使用, 而所有其他单元闲着。对于四路并行处理器, 如果 $N=4$ 和 $M=1$, 为什么并行的总体性能下降? 当 $M=0$, 新的功耗相比题中基准设计是多少?

2.4 在一个周期内 SRAM 单元的发生单比特软错误的泊松分布参数为 $\lambda = 10^{-25}$ 。假设只有一个字的 cache, 且在程序执行的第一个周期内数据进入该 cache。我们同时假设软错误仅发生在 cache, 在任何其他处理器的结构都不发生。

- (a) 在一个运行了 10 亿次循环的程序的执行过程中, 假设 cache 字在程序执行的 10 亿周期都是脆弱的, 那么在 cache 中具有单个位错误的概率是多少? 如果一个周期为 1ns, 缓存的 FIT 率 (以及在多年的 MTTF) 是多少? 注意, 这里计算出的 FIT 值称为本征 FIT 率。
- (b) 现在考虑每 100 万次循环之后的情况, cache 字在处理器中已经被新的数据所覆盖。在这种情况下, 这个 cache 的 FIT 率是多少? 需要注意的是, 当数据被刚刚进入 cache 新的值所覆盖, 此时任何旧值的软错误都是不相关的。
- (c) 现在考虑 cache 是由一位奇偶校验保护的情况。计算这个 cache 的 DUE 和 SDC 的 FIT 率。
- (d) 现在考虑 cache 是由一个单纠错双校验码 (SECDED) 保护的情况。计算这个 SECDED cache 的 DUE 和 SDC 的 FIT 率。
- (e) 现在考虑一个字有 2 位软错误位的概率为 30%, 即一个字出现 1 位错 70%, 同时 2 位错 30%, 当 cache 采用奇偶校验位和 SECDED 校验编码时, 比较 cache 的 DUE、SDC FIT 率。

2.5 正如本章所述, 电迁移 (EM) 是由无序电流引起, 测量 EM 影响的著名公式是 Black 公式:

$$\text{MTTF}_{\text{EM}} = A \times \omega \times J^{-n} \times e^{E_a/kT} \quad (2.25)$$

式中: A 是常数; ω 是横截面积; J 是电流密度; E_a 是活化能; k 是 Boltzmann 常数; T 是温度; n 是经验比例因子, 假设 $n=2$ 。

- (a) 如果线径加倍, 那么厚线的 MTTF 受 EM 影响有多少?
- (b) 如果线径加倍引发电路设计麻烦, 并因此导致温度提升 2 倍, 那么 MTTF 受 EM 影响后变为多少?

处理器微结构

3.1 概述

对于任何体系结构来说,处理器及其支持的指令集都是基本组成部分,因为它们从根本上决定了体系结构的功能特性。从某种意义上说,可以把处理器当作计算机系统的“大脑”,因此,理解处理器是如何工作的对于理解多处理器的工作原理至关重要。

本章首先介绍指令集,包括“异常”的概念。异常处理可以被看作是对处理器指令集的软件扩展,是完整的指令集架构定义的组成部分,也是必须支持的,它们反过来也会限制处理器体系结构的实现。如果不支持异常处理,处理器和多处理器可能会变得更高效,但是,也会在很多方面失去由软件对指令集扩展所带来的灵活性和便捷性。在本书中,我们使用一种非常类似 MIPS 的简单指令集作为基本指令集系统。之所以选取 MIPS,是因为它比较简单,基于它来解释处理器结构的基本概念会更容易理解。此外,我们还会根据需要介绍一些更加复杂的指令集,比如 Intel x86 指令集。

由于本书描述的是并行架构,所以我们不会介绍那些只能同时执行一条指令的简单架构。因此,本书将从五级流水线开始介绍,它在每个时钟周期内最多可以并行处理 5 条指令。在五级流水线中,指令的执行顺序(或者叫指令调度顺序)在编译阶段就已确定,这个顺序通常叫作程序序、线程序或者是进程序,并且硬件不对指令执行顺序进行动态调整,这种结构称为静态流水线。五级流水线包含一些基本的处理机制,如流水线阻塞、数据前递以及刷流水线等。这些处理机制是所有处理器体系结构都会使用的基本硬件机制,因此必须深入理解。通过扩展五级流水线,我们可以得到超流水处理器和超标量处理器。超流水处理器的主频比五级流水线高,在五级流水线中某一级完成的操作有可能被细分为多个更小的操作,并在超流水线中采用多个流水级来完成。此外,一些更加复杂的指令(如浮点运算指令)操作可以直接以流水的方式送入处理器的执行单元。与超流水线这种细化流水的方式不同,静态超标量处理器的扩展方式是在每个周期内同时读取或者执行多条指令。

静态流水线效率的提高主要依靠编译器优化。尽管编译器对代码十分熟悉并且可以很容易地识别出诸如循环这样的特征,但是编译器无法掌握某些需要硬件获取的动态信息,比如内存地址等。而支持动态调度的乱序(Out-of-Order, OoO)处理器既可以利用静态信息也可以利用动态信息,乱序处理器可以充分开发每个线程中的指令级并行(Instruction-Level Parallelism, ILP)程度。不过要想充分开发指令级并行并不是一件容易的事情,因为在指令乱序执行时,我们必须正确处理数据依赖、条件分支结果以及异常等情况。因此,我们需要一些相关的机制来确保内存和寄存器操作可能会带来的数据依赖,并且支持指令的推测执行。在推测执行中,指令会在分支条件结果得出之前就推测执行。为了支持指令推测执行,必须预测分支条件,且需要为预执行得出的结果提供临时存储区,以保证指令执行的程序序。当预测分支失败或者发生异常时,必须有相应的回滚机制来取消推测执行结果,并且重新开始执行正确指令。综上所述可以看出,动态流水线比静态流水线要复杂得多。

上述结构的复杂性可能会影响时钟频率,并且也会为在一个时钟周期内执行很多条指令的超标量动态流水线的设计带来困难。因此,处理器设计的另一个方面是充分利用编译器的优

化，同时简化硬件来保证机器的高主频。在超长指令字（Very Long Instruction Word，VLIW）处理器中，可以在不过度增加硬件复杂度的前提下，在每个时钟周期内处理大量指令。实际上，超长指令字处理器对硬件支持的要求非常低，所有关于操作依赖以及推测执行的问题都在编译阶段被静态处理。在向量处理机中，运算操作被编译器重新编译成向量操作，每条向量指令定义了大量相似但不相关的操作，这些操作使用大量的同类型标量操作数。因此，向量操作可以在很高的主频下实现很深的流水级。

本章涉及的内容如下：

- 3.2 节主要介绍指令集架构（ISA）。
- 3.3 节主要介绍乱序静态流水线的实现、超流水以及超标量静态流水机。
- 3.4 节主要介绍带推测功能的动态乱序流水线结构。这节将会讲述怎样处理内存和寄存器操作带来的依赖，怎样进行分支预测，怎样在条件分支中进行推测执行，最后将讲述怎样协调这些机制来保证程序的正确执行。
- 3.5 ~ 3.6 节主要介绍超长指令字机器及其编译器支持，以及显式并行指令计算（EPIC）结构。
- 3.7 节主要介绍向量机。

3.2 指令集架构

指令集架构（ISA）是计算机系统最基础的接口。它在软件设计者和硬件设计者之间定义了一个简单而又清晰的界限。软件设计者设计编译器以及操作系统，硬件设计者进行 ISA 的硬件实现。这两者是完全不同的群体，因此指令集架构必须以一种双方都可以理解的方式严格实现。ISA 将编译器、汇编代码或操作系统程序员所要进行的工作从物理层复杂的硬件中分离出来。反之亦然，硬件设计者按照 ISA 说明实现 ISA 时也不用关心应用于 ISA 上的复杂软件。

图 3-1 给出了现代计算机系统的层次化示意图，每一层都依赖于下一层。用高级语言（如 C++ 或 Fortran）编写的应用程序在用户层通过调用函数库来实现一些复杂或是常用的功能，或者进入系统层使用系统调用来实现操作系统提供的诸如 I/O 操作和内存管理等功能。编译器将代码编译到机器可以理解（目标代码）和操作系统可以理解（操作系统调用）的层次。操作系统可以通过软件扩展硬件的功能，通过软件实现复杂的功能，并且以一种高效、安全、透明的方式协调多用户共享系统资源。而在这些复杂的软件层下面就是 ISA 层。

应用程序
编译器/宏和程序库
操作系统
指令集（ISA）
计算机体系结构（组成）
电路（硬件功能实现）
半导体物理

图 3-1 计算机系统层次视图

ISA 将软硬件隔离开，ISA 的实现与其上的所有软件层都是独立的。计算机架构师的目标是在现有的技术条件下，设计出硬件结构来尽可能高效地实现指令集。由于架构师是在软硬件中间层进行设计，所以，他必须对两者都很熟悉。架构师在熟悉编译器与操作系统的同时，还要了解现有的技术限制。因此，可以看出，理解计算机体系结构对软件和硬件设计者来说都至关重要。

通过将硬件层从软件层分离出来，ISA 在计算机工业过去 50 年的巨大成功中扮演着关键性角色。在 20 世纪 50 年代至 60 年代初，每一台新计算机都有一套不同的指令集，指令集的设计在当时已经成为每台计算机设计的一部分。这样做的缺点是软件失去了在机器之间的可重用性（那时并没有编译器，所有的程序代码都由汇编语言编写）。1946 年，IBM 公司

通过推出 System/360 成为大名鼎鼎的计算机巨头。从那时起, IBM 保证其未来生产的所有计算机都可以运行 System/360 上的软件, 这些机器都将永久支持 IBM System/360 的指令集。这种被称为向后兼容 (backward compatibility) 的规定使得所有为 System/360 编写的代码都可以在任何一台 IBM 机器上运行, 从而省去了软件重写的代价。向后兼容是指当前的计算机可以运行之前任意一款机器上的二进制程序代码。这个保证是为了在未来的机器上支持所有的旧指令。IBM 360 指令集将来可能会有所扩展 (它也确实这么做了), 但绝不会舍弃已有指令或改变其语义。

虽然现在大多数程序都由高级语言编写并被编译成二进制文件, 但这种向后兼容的策略还是经受住了时间的考验。因为二进制文件的源代码可能丢失, 更重要的是, 通常软件供应商只提供二进制文件, 因此, 确保这些二进制文件可以在未来的任何机器上都能运行就显得非常重要。

总的来说, 指令负责将命令从软件转换到硬件, 它们是程序执行的基本步骤。指令由操作码 (opcode) 和操作数组成, 包括输入操作数和输出操作数。操作数可以存于内存中或 CPU 的寄存器中, 可以是显式的也可以是隐式的。当操作数隐含在操作码中时称为隐式的, 显式的则需要在操作数域显式指明。

3.2.1 指令类型和操作码

指令有固定的格式, 因此很容易通过硬件译码, 也有利于编译器生成。每一条指令都包含一个操作码 (命令) 以及数个输入/输出操作数 (数据)。

指令的操作码指明了这条指令需要硬件做什么操作, 本书中将会涉及 4 类指令:

- 整型算术/逻辑指令。
- 浮点算术指令。
- 访存指令。
- 控制指令。

算术/逻辑指令

整型算术指令对无符号整数或二进制补码进行算术操作。 n 位无符号整型可以表示 $0 \sim 2^n - 1$ 之间的所有正数和 0, 无符号整型最常用于地址表示。二进制补码可以表示 $-2^{n-2} \sim 2^{n-2} - 1$ 之间的正数和负数。对二进制补码操作数的典型操作有加 (ADD)、减 (SUB)、乘 (MULT) 以及对应的无符号数操作 (如 ADDU、SUBU、MULTU)。无符号数操作不会引起异常, 而有符号数操作在出现上溢或下溢时会出现异常。同一个算术逻辑单元 (Arithmetic Logic Unit, ALU) 可以执行有符号数操作和无符号数操作两种类型的指令。逻辑指令会对每个输入操作数进行按位操作, 典型的逻辑指令有或 (OR)、与 (AND)、非 (NOR) 以及与非 (NAND) 等, 逻辑指令也不会引起异常。

通常来讲, 大多数整型算术/逻辑指令可以在一个时钟周期内由一个 ALU 完成。对于乘除法指令来说, 由于其复杂性, 需要多个时钟周期才能完成。

浮点指令

浮点指令也是算术指令, 但其操作数由科学计数法表示 (包括符号 sign, 阶码 exponent 和尾数 mantissa)。这样做的目的是扩大可表示数的范围并且能够避免算术上溢或下溢。浮点指令被大量地用于科学和工程应用中, 并且必须在桌面 (工作站) 环境中支持。浮点数操作 (FADD、FMUL 和 FDIV) 要比整型操作复杂得多, 我们可以通过程序或宏来实现, 但如果通过硬件实现的话, 执行速度上还是要快得多。

访存指令

访存指令包括 load (LB、LH、LW 和 LD, 分别代表加载字节、加载半字、加载字、加载双字) 和 store (SB、SH、SW 和 SD, 分别代表存储字节、存储半字、存储字、存储双字) 两类。它们的作用是从内存中取数或向内存写数。指令包含一个内存地址并指定一个处理器内部的位置, 操作码显式地指明了内存操作数的大小。其他访存指令包括同步指令, 例如 test-and-set 指令或者 swap 指令, 同时也包括 load 和 store 操作。我们将在第 7 章对这些类型的指令进行讲解。

分支和跳转

指令按照程序计数器顺序执行。每个时钟周期程序计数器 (Program Counter, PC) 都会更新以指向顺序执行的下一条指令。控制指令 (分支和跳转) 打破了指令执行的顺序性。通常分支指令是有条件的, 当且仅当某一条件满足时才会进行分支。因此, 在执行分支指令时, 必须要计算出条件和目标地址。分支指令的目标地址包含在指令中, 是一个相对于分支指令 PC 值的二进制补码偏移。

不同 ISA 的分支指令之间的差别主要源于对分支条件的设置。在某些 ISA 中, 条件码主要由指令根据运算结果设置。举个例子, Motorola 68000 有 5 个条件码: Z (零)、C (进位)、V (溢出)、X (扩展) 和 N (负值), 有一个状态寄存器保存这些条件位并且作为程序状态字的一部分。条件码 (condition code, CC) 的主要问题包括: (1) 不管是不是真的需要, 所有指令都会对条件码进行设置; (2) 对条件码的设置取决于指令种类; (3) 分支指令和所有分支前的指令之间会引入额外的相关, 这将限制处理器对并行性的开发。

一种可以用来替代条件码的方法是设置条件标识, 只在需要时才检测它。这种方法使用指令对一些操作数进行测试, 并将结果存入某一寄存器中。例如:

```
SLT R1,R2,R3
BEZ R1,loop
```

在这个代码段中, SLT 指令在 $R2 < R3$ 时对 R1 置 1; 否则将 R1 置 0。BEZ 指令检测 R1 的值 (0 或 1), 并且当值为 0 时进行跳转。另一种方法是在分支指令中既计算条件又对其测试, 如果成功则跳转, 例如:

```
BNE R1,R2,loop
```

BNE 指令比较 R1 和 R2 的值, 如果不相等则跳转到 loop 处。

程序中经常会出现跳转指令 (有时也称为无条件分支), 跳转的地址通过指令本身给出 (直接跳转) 或由某个寄存器给出 (间接跳转)。与分支不同的是, 跳转指令的目标地址不需要计算。

通过跳转指令来实现子程序或程序调用并不高效, 因为必须保存返回地址 (函数调用后的下一条指令地址)。在 MIPS ISA 中, JAL (jump-and-link) 指令可以首先将返回 PC 值保存在一个寄存器中, 然后再跳转到子程序。在调用前或调用后, 原有上下文的寄存器值必须通过软件保存在内存的控制栈中。

```
JAL R28,subroutine
```

这条指令将下一条指令的 PC 值保存到 R28 寄存器中, 然后跳转至 Subroutine 子程序。

此外, 一些老的 ISA 有复杂的子程序调用指令, 可以完成所有上下文保存和现场恢复。然而, 这类复杂指令不够灵活, 并且子程序调用相对来说也不是很频繁, 根据 Amdahl 定律, 我们没有必要实现这种复杂性。

3.2.2 指令混合

熟悉各种指令/操作在一个程序中所占的比例是非常重要的，这种指令分布情况称为指令混合 (instruction mix)。指令混合分为两种：静态的或动态的。静态指令混合是指程序代码中的各指令类型比例；动态指令混合是指程序执行时各种指令类型的比例。在动态指令混合中，某个固定 PC 指向的指令可能会被执行多次（如果它是循环或子程序的一部分），而在静态指令混合中只会被记录一次。在指导 CPU 设计方面，动态指令混合比静态指令混合更有用。CPU 的设计应该关注动态指令混合中经常出现的指令，比如那些在一个实际程序执行期间耗时最多的指令。

表 3-1 给出了一个典型的动态指令混合的组成。load 指令的 CPI 很高是因为“内存墙”的问题，条件分支指令的 CPI 也很高是因为它们打破了顺序取指的可预测性，在分支条件计算出来之前，无法确定是不是应该跳过分支执行后续指令。这一问题对于一次只能执行一条指令的机器没有什么影响，然而，在更加复杂、流水化的机器中，条件分支指令会严重制约对代码并行性的开发。毫无疑问，处理器设计必须重点关注对 load 指令和条件分支指令的优化。此外，取指也是很重要的一个问题，因为所有的指令都要被取到处理器中，并且处理器没有指令时将无法执行。

表 3-1 整型 Benchmarks 中的动态指令混合粗略统计

操作码分类	比例 (%)	CPI
load 指令	25	高
store 指令	12	低
ALU 操作指令	40	低
条件分支指令	20	高
跳转指令	2	低
子程序调用指令	1	低

3.2.3 指令操作数

指令操作数保存在 CPU 或内存中。

CPU 内部操作数

CPU 内部的操作数可能保存在累加器、栈、寄存器或者是指令内部。

累加器。累加器是 CPU 内部隐含的一个操作数。累加寄存器作为操作数由如下指令的操作码隐式给出：

```
ADDA <mem_address>
MOVA <mem_address>
```

在 ADDA 指令里，存储在内存地址为 < mem_address > 的操作数会与累加器 A 内的数据相加，并且运算结果也保存在累加器中。在 MOVA 指令里，会将累加器 A 内的内容存储到内存中。CPU 内部可能有多个累加器 (A, B, …)。

堆栈。操作数存在于硬件构成的栈中。硬件堆栈由一组硬件寄存器组成，栈顶元素被默认为操作数，例如：

```
PUSH <mem_address>
ADD
POP <mem_address>
```

PUSH 指令将内存中地址为 mem_address 的数据取出,并放置在 CPU 内部堆栈的栈顶位置。ADD 指令将栈顶的前两个元素取出相加,并将结果存入栈顶。POP 指令将栈顶元素弹出,放在内存地址为 mem_address 的内存单元中。

寄存器。CPU 内部有一组可编址的寄存器,每个都有一个寄存器号。可以指定某些寄存器存储数据,另一些寄存器存储地址,例如:

```
LW R1,<mem_address>
ADD R2,<mem_address>
ADD R1,R2,R4
SW R10,<mem_address>
```

这些指令中的 R1、R2、R4 和 R10 都是 CPU 内部的可编址寄存器。寄存器相比于累加器或堆栈有自身的优势,它们的应用更加灵活,多个寄存器的值都可以重复利用。编译器对寄存器的使用进行管理,编译器尽可能重用存储在寄存器中的数据。当编译器需要寄存器分配一个新值时,编译器必须先将某一个寄存器的值写回到内存,即发射一条 store 指令将数存到内存中,从而释放该寄存器,这一过程叫作寄存器溢出 (register spill)。之后可能又需要这个数,此时需要重新分配一个寄存器,并通过 load 指令将数装入,这一过程叫作寄存器装入 (register fill)。很多现代 ISA 都只使用寄存器而不使用累加器或堆栈。然而,一般来说,给定的一个 ISA 可能使用累加器、堆栈和寄存器组中的一种或多种。

立即数。立即数是包含在指令内部的常数,将经常使用的较小的数作为常数可以获得很高的效率,例如:

```
ADDI R1,R2,#5
```

在这条指令中,特殊操作码 (ADDI) 表示加法的第三个操作数是一个立即数,且包含在指令本身的立即数域。在很多指令集中,寄存器 R0 的值恒为 0,可以作为操作中 0 的来源。

访存操作数

访存操作数由其有效地址 (effective address) 给出,指令内的访存操作数地址域会给出怎样计算每个访存操作数的有效地址。

通常,机器是按字节编址的,也就是说内存的每个字节都有其自己的地址,同样每个内存地址都会指向内存的某个字节。例如,一个 32 位的地址可以寻址 4GB 的内存空间。在某些 ISA 中 (尤其是现代的 ISA),操作数必须是对齐的 (aligned)。也就是说,一个操作数的地址必须是它大小的整数倍。因此,半字 (2 个字节) 的地址必须是偶数,字 (4 个字节) 的地址必须是 4 的倍数,而双字 (8 个字节) 的地址必须是 8 的倍数。这样限制的目的是简化与数据 cache 的接口,因为 cache 块的大小通常是 2 的幂,整个操作数必须被包含在同一个 cache 块以及同一个内存页中。缺页或 cache 失效是不允许在访问操作数或指令过程中发生的。编译器或汇编代码要保证操作数对齐。若操作数没有对齐,硬件则无法处理本次访问,此时会触发一个故障 (异常)。

关于字节可编址性的另一个敏感问题是字节的高位是随着地址增加还是减小。也就是说,内存中的最高有效字节 (Most Significant Byte, MSB) 应该放在地址最低的地方还是最高的地方,这两种方式分别称为大端 (big endian) 和小端 (little endian)。为了说明这个问题,图 3-2 给出了内存中的第一个字 (地址 0 单元) 的示意图,一个字包含 0,1,2,3 四个字节。在大端方式中,MSB 是字节 0,最低有效字节 (Least Significant Byte, LSB) 是字节 3;在小端方式中,MSB 是字节 3,LSB 是字节 0。

这种规定看上去无太大意义,然而,这种约定甚至对程序源码级的输出都会产生影响。如果代码获取按字打包的字节数据,比如操作 ASCII 编码文本的程序,程序中某些处理打包成字

的字节数据的代码可能在采用不同大、小端方式的体系结构之间就不能直接移植了。例如，加载 0 号字节的結果可能是获取到 0 号字的 MSB（大端）或 0 号字的 LSB（小端）。

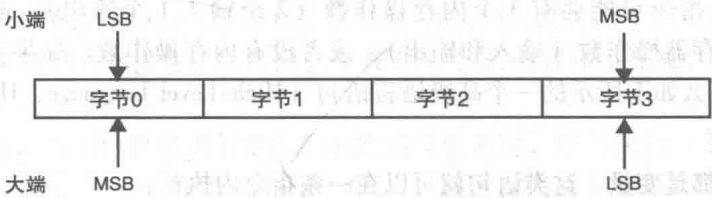


图 3-2 大端与小端存储格式

Sun 公司的 SPARC 微系统和 IBM 公司 System 370 的 ISA 采用大端方式，而 Intel x86 采用小端方式。在某些 ISA（例如 MIPS）中，采取哪种方式可以在启动系统时通过某一模式位设定。

寻址方式

指令中的操作数地址域控制硬件通过地址寻址模式加所需的一些数据域值形成有效地址，然后根据有效地址去相应的地址空间取操作数。这些数据域保存了用来计算地址的所需的数据（如寄存器号和偏移地址）。有些情况下，寻址方式被隐式地编码在操作码中。表 3-2 给出了一些寻址方式。有些指令集支持多种寻址方式。在只基于寄存器的 ISA 中有三种寻址方式：指定 CPU 操作数；指定内存操作数的地址；指定分支目标指令地址。

表 3-2 基于寄存器的机器寻址示例，(Ri) 表示 Ri 中的值，MEM [x] 表示内存地址 x 中的值

寻址模式	示例	含义
寄存器直接寻址	ADD R1, R2, R3	$R1 \leftarrow (R2) + (R3)$
立即数寻址	ADDI R1, R2, #15	$R1 \leftarrow (R2) + 15$
偏移寻址	LOAD R1, #20(R2)	$R1 \leftarrow \text{MEM}[(R2) + 20]$
寄存器间接寻址	LOAD R1, (R2)	$R1 \leftarrow \text{MEM}[(R2)]$
内存直接寻址	LOAD R1, #2000	$R1 \leftarrow \text{MEM}[2000]$
内存间接寻址	LOAD R1, @(R2)	$R1 \leftarrow \text{MEM}[\text{MEM}[(R2)]]$
取值后自加	LOAD R1, (R2) +	$R1 \leftarrow \text{MEM}[(R2)]$ then $R2 \leftarrow (R2) + \text{size}$
取值前自减	LOAD R1, - (R2)	$R2 \leftarrow (R2) - \text{size}$ then $R1 \leftarrow \text{MEM}[(R2)]$
PC 相对寻址	BEZ R1, #166	$\text{PC} \leftarrow (\text{PC}) + 4 + 166$

一些内存寻址方式是其他寻址方式的特例。例如，寄存器间接寻址和内存直接寻址都是偏移寻址的特殊情况。其他寻址方式可以通过一些简单寻址方式的组合而得到。例如内存间接寻址，取值后自加和取值前自减可以通过两条指令简单合成：

```
LOAD R1,@(R2) ==>    LOAD R3,0(R2)    /内存间接取值
                      LOAD R1,0(R3)
LOAD R1,(R2)+ ==>    LOAD R1,0(R2)
                      ADDI R2,#size    /取值后自加
```

如果 16 位的偏移量或立即数不够用，32 位的地址或值可以先放在寄存器中再使用。注意在组合寻址方式中，编译器可能会多使用一个额外的寄存器，例如上面内存间接寻址例子中的 R3。另一方面，编译器可能会丢失地址寄存器的初始值（比如上例中重用 R2 而不是使用 R3），这可能会导致寄存器溢出和装入。不过，在实际程序中，这些特殊的寻址方式出现的频率很低，根据 Amdahl 定律，它们对程序性能的影响都可以忽略。

内存操作数的个数

基于寄存器的 ISA 可以根据操作数的个数和 ALU 指令中内存操作数的个数进行分类。在某些机器中, ALU 指令可能会有 3 个内存操作数 (2 个输入 1 个输出), 或 1 个内存操作数 (输入) 和 1 个寄存器操作数 (输入和输出), 或者没有内存操作数。如果一个 ALU 指令有 3 个内存操作数, 那么如下所示的一个高级语言语句 (High-Level Language, HLL):

```
C = A + B
```

其中, A、B 和 C 都是变量, 这类语句就可以在一条指令内执行:

```
ADD C, A, B
```

其中, A、B 和 C 是内存操作数地址, 也就是说一条指令对应于一条程序语句。这种指令有两个缺点: 一是指令长度太长 (表明内存操作数所用的位数远大于寄存器所用的位数); 二是大量的内存读写, 因为这些操作数总是从内存中读取, 从不会在 CPU 内部暂存和重用。

上述 HLL 语句通过一个内存操作数和一个寄存器操作数可以被编译成下面的格式:

```
LOAD R1, A
```

```
ADD R1, B
```

```
STORE C, R1
```

显然, 这个代码长度肯定会比 3 个内存操作数的代码长度长。然而, 一个主要的优点是, C 的值被保存在 R1 里, 如果以后需要的话就不用再从内存中读取了。此外, 如果 A 之前就被存储在一个寄存器内, 那么 LOAD 指令也不再需要了。

基于寄存器寻址的 ISA 中, ALU 指令里不包含内存操作数, 这种 ISA 也称为 load/store ISA。在这种 ISA 中, 所有的操作数都在执行 ALU 操作之前必须显式加载到寄存器中。上述 HLL 语句在 load/store ISA 中被编译成如下:

```
LOAD R1, A
```

```
LOAD R2, B
```

```
ADD R3, R1, R2
```

```
STORE C, R3
```

正如之前所说, 这个代码序列会比 3 个内存操作数的指令更长, 并且会用掉更多的寄存器。但是, 与上面讲到的类似, R1、R2、R3 的内容可能在以后被重用, 另外, A、B 可能已经存在某个寄存器中了, 从而可以省掉两个指令。load/store ISA 的主要优点是对硬件的简化, 尤其是对指令译码和流水指令的简化。

3.2.4 异常、陷阱和中断

异常 (exception) 正如其字面意思, 是一类非常罕见的由硬件触发并强制处理器执行异常处理的事件。和分支或跳转相比, 异常最主要的不同就是它是由硬件触发而不是在程序代码中主动调用, 当然有时候这个差异并不明显, 比如有些异常是由内核陷阱引起的。由于它们比较相似, 因此在绝大多数机器中, 分支处理和异常处理共用同样的硬件机制。异常机制是 ISA 规范的一部分, 所有硬件实现都需要支持。

异常可能是由指令、外部中断或硬件故障引起的。由程序指令引起的异常需要保持和指令的同步。如果程序要在异常处理后继续执行, 就必须停在出错的指令处并重新执行该出错指令。与之相反, 由 I/O 中断和硬件故障/宕机引起的异常不需要和程序指令保持同步, 它们可能在程序的任何一条指令上产生中断。不过, 中断还是需要及时处理, 以免丢失。由于中断同程序指令无需同步, 因此总体上更容易处理。接下来, 我们讨论几种异常的例子。

I/O 设备中断。绝大多数 I/O 操作由于延迟很长, 只能由中断驱动。通常, CPU 通过直接

内存访问 (Direct Memory Access, DMA) 控制器或 I/O 处理器等 I/O 设备来执行一次 I/O 操作。然后, 在 I/O 设备执行 I/O 操作时, CPU 会继续执行当前的程序。当 I/O 操作完成时, I/O 设备会通过中断信号告诉 CPU。当 CPU 接收到信号时, 它会执行一段中断处理程序。因为 I/O 设备对于 CPU 来说是外部设备, 并且同正在被 CPU 执行的程序不相关, CPU 可以选择在任意时钟周期来处理此中断。

操作系统调用。当用户想要调用操作系统的某项服务时, 它会执行一条陷阱 (TRAP) 指令。通常 TRAP 指令有一个参数指向 Trap 表的入口, 通过查找这个表可以得到请求服务处理程序的入口地址。这种异常处理同跳转到子程序的操作非常相似。

指令追踪和断点。绝大多数机器提供对程序执行的追踪。在追踪模式中, CPU 在执行每条指令时都陷入 TRAP, 以便异常处理程序可以在指令执行之前先将 CPU 状态记录到 trace, trace 通常存储在磁盘上。有了追踪机制, 可以得到所有有用的信息, 从而为编译器和体系结构设计提供帮助。记录的 trace 可以用来评价其他的体系结构。此外, 与之相似, 硬件也通过异常处理支持用于程序调试的断点。

整型或浮点异常。算术指令 (尤其是浮点运算) 可能会导致一系列的异常, 最常见的是下溢出异常和上溢出异常。若碰到这两种异常处理, 可能中止进程也可能继续。算术异常也可能表示某些非法操作, 比如除零操作。

缺页。现代系统一般都支持虚拟内存。系统的物理内存只包含进程所需的很少一部分指令/数据页。如果在内存的某一页上没有找到要取的指令或数据, CPU 就会转到核心态来执行内存管理函数, 把缺少的页读到内存中。

非对齐访存。当数据没有根据其大小在内存中对齐时, 就会出现非对齐访存异常。

违反内存保护。访存可能越界或超越访问权限 (例如, 对只读页执行写操作)。这种情况下, 程序必须在造成错误前及时终止。

未定义指令。当指令译码器遇到了未知/非法代码 (操作码或寻址模式) 时, 它不知道如何处理, 此时必须进入异常。这种异常可以用于扩展 ISA, 例如, 一个没有浮点操作的指令集, 可以将非法操作码分配给浮点指令并在异常处理程序中处理。

硬件故障/警报。各种硬件组成部分, 如总线和内存等, 通常都有错误检测/纠正码等机制保护。当某个硬件组成部分无法从错误中恢复时, 就必须让正在 CPU 上执行的进程陷入 trap。在现代计算机系统中, 整个机器中分布了各种传感器, 当某些条件 (比如局部温度) 进入危险区域时可能会让 CPU 陷入 trap。

掉电。掉电时, 由于电容的作用, 电压会缓慢下降。因此就有时间在系统停止之前尽可能保存正在进行的计算和环境, 以便后续进行恢复。

当某个异常需要和指令 i 保持同步, 并且进程在异常处理完之后需要恢复运行时, 那么必须保存第 $i-1$ 条指令结束后和第 i 条指令开始前的处理器状态, 从而使得处理器可以在处理完异常之后恢复到第 i 条指令执行。因此所有按照进程序在指令 i 之前的指令都必须执行完, 而包括指令 i 在内的所有后续指令都不能执行。如果每次只执行一条指令, 那么进程序就是指令执行的动态顺序。这种异常叫作精确异常。例如, 缺页或算术溢出等异常必须是精确的, 而硬件错误或访存冲突等其他异常并不需要精确处理, 因为在异常处理结束后进程会被终止。

精确异常对硬件或编译器可以做的优化产生了限制。只有程序员设置的异常才可能被触发。尽管异常十分少, 但是可能由任意指令产生, 并且在指令到达执行阶段之前是无法预测的, 因为异常是在执行阶段被检测到的。由于异常很少, 没有必要加速其处理过程。然而, 编译器和体系结构必须有内在的机制可以检测精确异常并从中恢复。

3.2.5 存储一致性模型

存储一致性模型是 ISA 至关重要的一个组成部分，它定义了多核处理器中多进程或多线程访问时的合法交错顺序，我们将在第 7 章详细介绍这一问题。

3.2.6 本书的核心 ISA

本书中绝大多数例子和习题使用的核心 ISA 是 MIPS ISA 的一部分，为了方便概念解释，指令集被精简到最小状态，我们会根据需要介绍更多的指令。

指令和操作数类型

本书提到的 ISA 都是 load/store 类型，所有 ALU 指令的操作数取自寄存器并且含有 3 个操作数。load 和 store 指令将数据从内存取到寄存器或从寄存器存储到内存中，所有的指令被编码成 32 位（4 字节）的字长度对齐存到内存中。

整型数据类型有字节、半字（2 字节）、字（4 字节）和双字（8 字节）这几类。浮点数据类型有单精度（4 字节）和双精度（8 字节）两类。所有的操作数都在内存中对齐，并且其大小端方式可以配置。有两个分离的寄存器组：一组是整数寄存器（R0 ~ R31）；一组是浮点寄存器（F0 ~ F31）。双精度浮点操作数（8 字节）占据两个连续的寄存器，并且寄存器号是偶数。R0 被置为 0，用来提供常数 0。

寻址方式有寄存器直接寻址（所有 ALU 指令和浮点指令）、偏移寻址（内存操作数）、立即数寻址、PC 相对寻址（分支指令）、绝对内存寻址和寄存器间接寻址（跳转）。本书中用到的核心 ISA 的各种指令在表 3-3 中给出。

表 3-3 本书的核心 ISA

指令类型	操作码	汇编码	含义	备注
数据传输指令	LB, LH, LW, LD	LW R1, #20(R2)	$R1 \leq \text{MEM}[(R2) + 20]$	针对字节、半字、字、双字长度
	SB, SH, SW, SD	SW R1, #20(R2)	$\text{MEM}[(R2) + 20] \leq (R1)$	
	L.S, L.D	L.S F0, #20(R2)	$F0 \leq \text{MEM}[(R2) + 20]$	单精度/双精度浮点取数
	S.S, S.D	S.S F0, #20(R2)	$\text{MEM}[(R2) + 20] \leq (F0)$	单精度/双精度浮点存数
ALU 操作指令	ADD, SUB, ADDU, SUBU	ADD R1, R2, R3	$R1 \leq (R2) + (R3)$	有符号/无符号加法/减法
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1, R2, #3	$R1 \leq (R2) + 3$	有符号/无符号加立即数/减立即数
	AND, OR, XOR	AND R1, R2, R3	$R1 \leq (R2) \cdot \text{AND} (R3)$	按位逻辑与、或、异或
	ANDI, ORI, XORI	ANDI R1, R2, #4	$R1 \leq (R2) \cdot \text{ANDI} 4$	按位与、或、异或立即数
	SLT, SLTU	SLT R1, R2, R3	$R1 \leq 1 \text{ if } R2 < R3$ $\text{else } R1 \leq 0$	有符号或无符号比较，比较 R2 和 R3，结果输出到 R1
	SLTI, SLTIU	SLTI R1, R2, #4	$R1 \leq 1 \text{ if } R2 < 4$ $\text{else } R1 \leq 0$	有符号或无符号比较，比较 R2 的值，结果输出到 R1

(续)				
指令类型	操作码	汇编码	含义	备注
分支/跳转指令	BEQZ, BNEZ	BEQZ R1, label	$PC \leq \text{label if } (R1) = 0$	条件分支：等于 0/不等于 0 则跳转
	BEQ, BNE	BNE R1, R2, label	$PC \leq \text{label if } (R1) = (R2)$	条件分支：等于/不等于
	J	J target	$PC \leq \text{target}$	跳转目标在立即数域
	JR	JR R1	$PC \leq (R1)$	跳转目标在寄存器中
	JAL	JAL target	$R1 \leq (PC) + 4;$ $PC \leq \text{target}$	保存返回地址到 R31 中后, 跳转到目标地址
浮点指令	ADD. S, SUB. S, MUL. S, DIV. S	ADD. S F1, F2, F3	$F1 \leq (F2) + (F3)$	单精度浮点计算
	ADD. D, SUB. D, MUL. D, DIV. D	ADD. D F0, F2, F4	$F0 \leq (F2) + (F4)$	双精度浮点计算

指令格式

程序由汇编代码或高级程序语言编写，最终被翻译成机器所能识别的目标代码或二进制代码。指令中各部分到二进制的编码叫作指令格式。在本书的类 MIPS 指令集中，所有指令都是等长的 32 位，主要有三种格式，如图 3-3 所示。

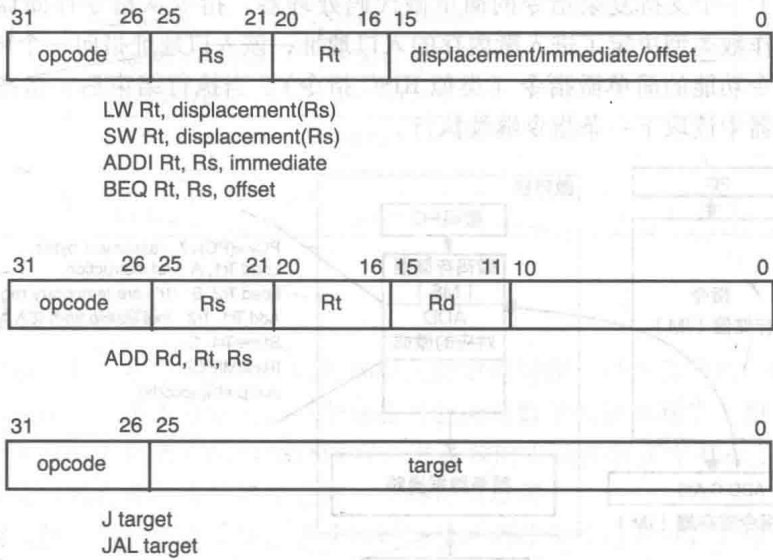


图 3-3 指令格式

3.2.7 CISC 和 RISC

现代指令系统（如 MIPS ISA）称作 RISC ISA，RISC 是指“精简指令集计算机”。RISC ISA 背后的基本哲学是从零开始构造指令集，最开始只包含少量的基本指令，之后逐步加入那些对性能有帮助的指令。为了简化译码、提升流水效率和缩短时钟周期，指令长度通常是固定的，格式也是规定好的，并且每条指令的执行也控制在一个周期内。

与之相反，传统的 ISA（如 IBM 370 系统或 Intel X86）则称作 CISC ISA，代表“复杂指令集计算机”。这类结构的一个主要目标是在可实现的基础上，尽可能多地实现指令编码，这就导致了指令的长度和格式不固定。其主要观点是，对于一段代码，指令流越紧凑，指令内存和

cache 的利用率就越高, 取的指令字节就越少。另外, 这类指令集出现的时候, 大多数程序都是由汇编代码编写的。当时的趋势是 (现在看来可能是错误的趋势), 应该增加指令集功能的复杂性, 以此来减轻程序员的负担。最终, CISC ISA 发展越久就越复杂, 时至今日, 人们还在不断地尝试向指令集中增加越来越多的功能。

DEC Vax-11 ISA (现在已经停用了) 曾是一个非常著名的复杂 ISA, 并且在 20 世纪 70 年代到 80 年代中期非常成功 (那时候, 程序的运行速度还在 1MIPS 水平!), 当时工作站、个人电脑以及 RISC 的观点开始出现。然而, Intel 的 iAPX432 指令集仍然在 20 世纪 80 年代初把复杂指令集推向了巅峰。iAPX432 是一个真正的超级 CISC, 它的指令集用硬件 (微代码) 直接实现了面向对象的处理, 指令集有 200 多个操作码, 指令长度从 6 位到 300 多位不等, 而且用哈夫曼编码进行编码。除了普通的整型和浮点操作数, ISA 可以操作的基本数据类型还包括数组、比特流和“对象”。iAPX432 被设计成用来在硬件层面支持面向对象的编程方法, 服务于 Ada 编程语言。这也同当时广泛流传的 ISA 设计要针对特定语言这一根深蒂固的观点相一致, 这种观点认为, 这样做可以把每个指令设计得尽可能接近高级语言的语句。这无疑导致了某些指令的极度复杂, 并且只可能通过微代码去实现。

复杂 ISA 必须通过微代码来实现, 因为直接用硬件实现复杂指令实在是太复杂了。微指令是一种十分简单的指令, 当复杂指令被取出并译码后, 这条指令的执行就由执行一串从微内存 (microstore) 中取出的微指令所代替。

图 3-4 展示了一个支持复杂指令的简单微代码处理器。指令从指令存储器中取出并被译码。操作码和操作数类型决定了进入微内存的入口地址, 该入口地址指向一个微程序, 其包含一串完成复杂指令功能的简单微指令 (类似 RISC 指令)。当执行结束后, 微指令序列使得处理器从指令存储器中读取下一条指令继续执行。

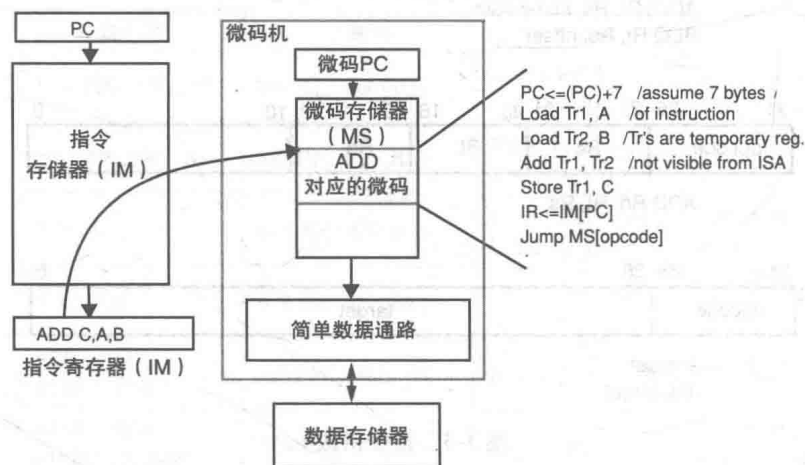


图 3-4 使用微码实现的机器

虽然这种个机器非常简单, 但仍旧非常慢。过去也有一些对伪代码进行并行化的尝试, 从而可以在一个时钟周期内执行多个微操作, 但这个方法并不十分有效, 因为一条指令内部的操作基本是顺序的, 可并行度很低。微指令存储空间中每个微字 (microword) 中只有一条简单的微操作 (如图 3-4 所示) 的情形叫作“垂直微代码”, 而每个微字有多个微操作的则叫作“水平微代码”。

反观 RISC 我们可以看到, 它消除了微指令系统的指令翻译开销, RISC 将源代码直接编译到微代码。也就是说, RISC 指令将微代码暴露给了编译器, 以便对其进行优化。微指令编码

机器的一个主要失败在于，无论指令复杂与否，都引入了额外的翻译开销。

必须强调的是，RISC 和 CISC 本身不代表硬件的复杂与否，它们指的是 ISA 的简单程度。正如刚才看到的，CISC ISA 可以通过简单的微码机来实现。同样我们会看到，RISC 机器随着时间的推移也变得越来越复杂。当然，可以预料的是，对于一种给定的实现方案，CISC ISA 的硬件系统会比 RISC ISA 的复杂。然而，由于今天的处理器已经十分复杂，实现 CISC ISA 的复杂性与实现 RISC ISA 的复杂性相比，已经越来越可以忽略不计了。基本上，复杂指令可以翻译为由简单指令组成的微代码，而微代码的执行就可以使用 RISC ISA 相同的机制了。

表 3-4 给出了一些主要 ISA 的例子。需注意的是 ISA 可能会有多种版本，例如，IBM System 360 后来逐步演化到 System 370、System 390，并一直到 System z。同样，SPARC 也随时间的变化出现了多个版本。当然，Intel x86 ISA 也在不断变化，其指令数从 20 世纪 60 年代开始一直在急剧增长。然而，后续版本必须保证向前兼容，这样才能确保软件的投入是值得的。注意有必要区分 ISA 和对应的实现（机器模型号或名称），因为每个 ISA 版本可能都会有多种不同的实现。

表 3-4 重要的 ISA 及其实现例子

ISA	公司	实现例子	类型
System 370	IBM	IBM 370/3081	CISC-legacy
x86	Intel	Intel 386, Intel Pentium IV, AMD Turion	CISC-legacy
Motorola 68000	Motorola	Motorola 68020	CISC-legacy
Sun SPARC	Sun Microsystems	SPARC T2	RISC
PowerPC	IBM/Motorola	PowerPC 601	RISC
Alpha	DEC/Compaq/HP	Alpha	RISC
MIPS	MIPS/SGI	MIPS 10000	RISC
IA-64	Intel	Itanium-2	RISC

3.3 静态调度流水线

过去为了有利于复杂指令的实现，机器都会支持微码操作。对于简单指令可以使其执行过程流水化。流水线在工业中十分常见，一个物品可能会由数千人流水制作。相对于由大量工人同时制作一个物品且每人负责不同的部分而言，更高效的方法是将多个物品通过装配线流动。这样一来，工人或机器在每一步只用完成某一个特定的步骤。

要实现流水线的机制，所有工作需要按照同样的或者非常接近的方式完成。指令执行是一种非常适合流水化的操作，因为大量指令是逐个执行的，并且绝大多数指令都要经过取指、译码、执行和写结果这些阶段。不过，为了提高效率，应该将不同指令类型在格式和执行上的差异最小化。因此 RISC ISA 十分适合流水线（尽管借助于将复杂指令转换成垂直微代码的形式，CISC ISA 也可以转换成 RISC 指令序列，以后会详述）。

指令流水的一个主要问题是指令执行不独立。通常，一条指令的执行需要得到前一条指令的执行结果再执行。有时，由于分支的原因，指令流水线甚至都不知道下一条该被执行的指令是哪条。此外，指令可能会导致异常，发生这种情况时，需要执行其他指令序列，此时当前任务需要被暂停，甚至是完全放弃。

在本节中，我们探索用于流水线和 RISC 指令并行化的体系结构技术，使用的指令集如表 3-3 所示。我们将从经典的五级流水线开始，这种简单结构包含了其他复杂机器中的基本硬件机制。

3.3.1 经典五级流水线

经典的 RISC 流水线包含 5 个阶段：取指（IF）、译码（ID）、执行（EX）、访存（ME）和写回（WB）。由于所有的指令都是定长的，因此 IF 阶段的动作对所有指令都是相同的。在 ID 阶段，对每条指令进行译码并且通常要读取两个寄存器。如果该指令是分支指令，则在此阶段计算目标转移的地址。EX 阶段用来计算地址或数据的值。对于分支指令，EX 阶段比较两个寄存器的值并且在满足条件后执行跳转。ME 阶段对于 load 和 store 指令有效，其他指令在该流水级什么也不做。最后，输出寄存器的值在 WB 阶段按要求进行更新。需要注意的是，即使指令在某个阶段没有任何动作，它也仍然要流经这个阶段从而使所有指令可以按照处理顺序流经流水线的每一个阶段。

表 3-5 给出了不同种类指令流经流水线的动作。由于浮点指令和整形乘除指令的执行阶段不止一个周期，它们不适合五级流水线，因此这两类指令并没有包括在表中，这些指令必须通过子程序来实现。

表 3-5 指令在流水线每一级中的活动

指令	IF 流水级	ID 流水级	EX 流水级	ME 流水级	WB 流水级
LW R1, #20(R2)	取指，且 PC + 4	译码，读取 R2 寄存器值	计算地址：R2 + 20	read	写入 R1
SW R1, #20(R2)	取指，且 PC + 4	译码，读取 R1 和 R2 寄存器值	计算地址：R2 + 20	write	—
ADD R1, R2, R3	取指，且 PC + 4	译码，读取 R2 和 R3 寄存器值	计算 R2 + R3	—	写入 R1
ADDI R1, R2, imm.	取指，且 PC + 4	译码，读取 R2 寄存器值	计算 R2 + imm	—	写入 R1
BEQ R1, R2, offset	取指，且 PC + 4	译码，读取 R1 和 R2 的寄存器值，计算目标地址：PC + offset	计算 R1 - R2，如果结果为 0，则分支跳转	—	—
J target	取指，且 PC + 4	译码，跳转到目标地址	—	—	—

在五级流水线中，指令在 IF 和 ME 两级都会访问内存。五级流水线没有 cache 失效的处理机制，因此，每一次 cache 失效，CPU 将暂停，直到 cache 失效处理完成后重新开始。

支持独立 load、store 以及 ALU 指令的基本五级流水线

图 3-5 给出了基本的五级流水线示意图，该流水线可以执行相互独立的 load、store 和 ALU 指令。相互独立的指令之间不共享寄存器或内存位置等资源。数据通路上的主要资源包括：指令存储（cache），寄存器文件（有两个读端口和一个写端口），可以进行整型算术和逻辑操作的 ALU，以及数据存储（cache）。

两个连续的流水级被流水线寄存器隔开，本节中，我们根据其隔开的两个流水级来对寄存器进行标记。当指令从一个流水级到另一个流水级时会进行重编码，重编码后的指令会被存储到流水线寄存器中。所有的流水线寄存器在每个时钟周期都可以操作。在每个时钟周期内，流水级执行如下操作。

取指（IF）。在每一个时钟周期（又叫一拍）中，程序计数器（PC）会在当前指令被取到指令寄存器中后加 4。在该拍结束时（这一拍的后沿），(PC) + 4 被存储到 PC 中，新指令保存到 IF/ID 寄存器中。

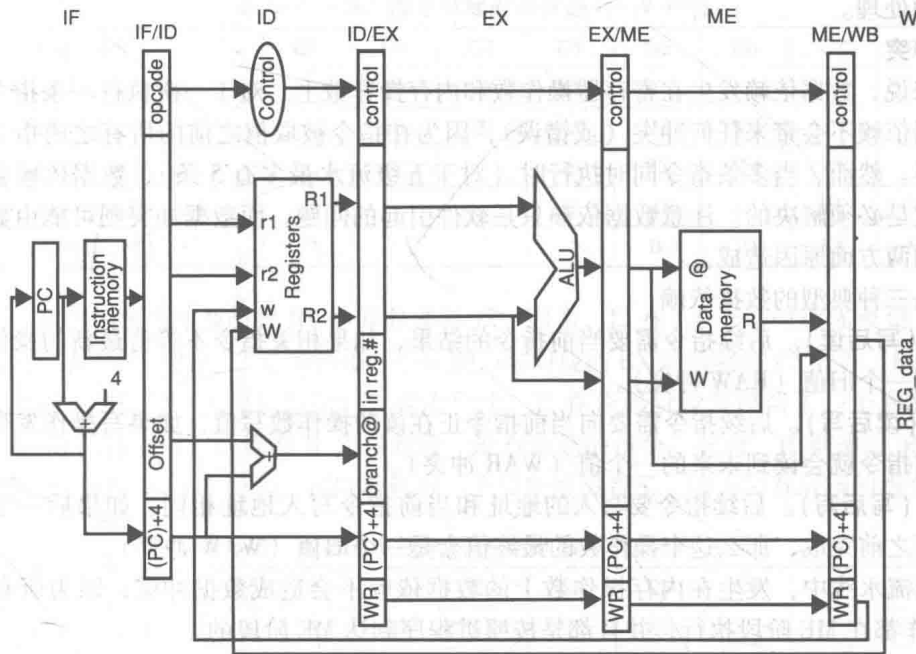


图 3-5 支持独立 load、store 和寄存器操作指令的五级流水线示意图

译码 (ID)。操作码被译码成控制信号，控制信号对之后连续的几个阶段 (EX, ME, WB) 建立各种操作的组合，并且同每一阶段的硬件组成部分的控制输入相联系。在时钟周期结束时，这些控制信号被存储在 ID/EX 寄存器的控制域中。本阶段还需要从寄存器文件中读取两个输入寄存器，尽管可能实际用不到读取的值。除了操作码之外的指令流入下一阶段 (EX)，(PC) + 4 必须被携带着在流水线中流动，因为当出现异常时需要用到 PC 信息。

执行 (EX)。对应于 EX 阶段的控制信号在此阶段执行，并且从控制域中提取出来。供 ME 和 WB 阶段使用的控制信号则继续传输保存到 EX/ME 寄存器中。ALU 的高位输入从一个输入寄存器中取值，ALU 的低位输入则可能连接到第二个输入寄存器（对于所有操作数都是寄存器操作数的 ALU 指令而言），或者（不会显示）连接到指令的低 16 位（用于 load 和 store 指令的地址计算，或 ALU 指令的 16 位立即数）。目的寄存器 (WR) 号通过 EX/ME 寄存器中的 WR 域被传输到下一阶段 (ME)。对于 load 和 store 指令，ALU 用来计算地址值。对于 store 指令，要存储的值通过旁路绕过 ALU 传输到下一阶段。

访存 (ME)。对应于访存阶段的控制信号在此阶段被执行，并且从控制域中提取出来。对应于 WB 阶段的控制信号则继续传输到 ME/WB 寄存器。对于 load 或 store 指令，ALU 的输出即为地址，我们将其放到内存的地址总线上。对于 store，从 EX 传输来的要存储的值连接到内存输入数据总线上。在这个周期的后沿会将值写到内存。对于 ALU 指令，该阶段什么也不做，ALU 的计算结果被直接送到 ME/WB 中。注意输出寄存器号也会被传输到 WB 阶段，保存在 ME/WB 的 WR 域。

写回 (WB)。剩下的控制信号都在本级进行处理，来自内存的值（对于 load）或者来自 ALU 的值（对于 ALU 指令）将存储在由 WR 域指明的寄存器中。注意这个寄存器的值在这一拍要被修改，而此时从该寄存器读出的值仍是旧值，这是因为寄存器值的更新在这一拍的后沿才会生效。

当指令在流水级之间流动时，它需要携带后续流水级需要的各种信息，比如控制信号、数据/地址信息、目的寄存器号等。当某些域不再需要时，就可以直接丢弃。这就是通常的流水线设计方法。在流水线中，指令只携带它们需要的信息。图 3-5 中的流水线没有对分支指令中

控制依赖的处理。

数据冲突

一般来说，数据依赖发生在寄存器操作数和内存操作数上。对于一次执行一条指令的机器来说，数据依赖不会带来任何冲突（或错误），因为在指令被取值之前的所有之前指令的结果都已经获得。然而，当多条指令同时执行时（对于五级流水最多有 5 条），数据依赖会导致数据冲突，这是必须解决的。注意数据依赖只是软件引起的问题，而数据冲突则可能由数据依赖以及微结构两方面原因造成。

下面是三种典型的数据依赖：

RAW（写后读）。后续指令需要当前指令的结果，如果相关指令不等待最新的操作数，它可能会读到一个旧值（RAW 冲突）。

WAR（读后写）。后续指令需要向当前指令正在读的操作数写值，如果写操作发生在读操作前，当前指令就会读到未来的一个值（WAR 冲突）。

WAW（写后写）。后续指令要写入的地址和当前指令写入地址相同。如果后一个写操作在第一个写之前完成，那么这个操作数的最终值会是一个旧值（WAW 冲突）。

在五级流水线中，发生在内存操作数上的数据依赖不会造成数据冲突，因为所有的 Load 和 Store 操作都在 ME 阶段执行，并且都是按照程序到达 ME 阶段的。

寄存器之间也不会发生 WAW 或者 WAR 冲突。WAW 冲突不可能，是因为所有的指令是在按进程序到达 WB 段后才更新寄存器文件的。对于 WAR 也是同理，指令总是先于进程中的后续指令到达译码阶段，并完成寄存器读取操作。

因此，还剩下寄存器间的 RAW 冲突。这种冲突是真实存在的，如表 3-6 所示。表中每一行表示一条指令从一个阶段到下一阶段的过程，按照图 3-5 的流水线一拍一拍执行。指令 I2、I3、I4 和 I5 都依赖于有 I1 产生的 R1 的值。以现在这个设计来看，唯一没有冲突的是 I5，因为 I1 将值写到 R1 是在 C5 的后沿，而 I5 是在 C6 到达 ID 阶段才读取值的。然而，目前来看，I2、I3 和 I4 仍然会读到 R1 中的旧值。

表 3-6 ALU 指令中的寄存器 RAW 冲突

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1, R2, R3	IF	ID	EX	ME	WB				
I2	ADDI R3, R1, #4		IF	ID	EX	ME	WB			
I3	LW R5, 0 (R1)			IF	ID	EX	ME	WB		
I4	ORI R6, R1, #20				IF	ID	EX	ME	WB	
I5	SUBI R1, R1, R7					IF	ID	EX	ME	WB

I4 执行过程中出现的关于 R1 的冲突可以通过修改寄存器来避免：如果一个寄存器在同一周期既被读又被写，那么被写的值可以直接送给读指令，这种优化叫作寄存器前递，其实现方式很简单，只需在寄存器文件的端口加几个多路复用器就可以了。

为了解决 I2 和 I3 关于 R1 的冲突，R1 的值在 C3 完成后就被计算出来了，而 I2 和 I3 分别在 C4 和 C5 的开始需要这个值。在 C4 的开始，R1 的新值刚被存储到 EX/ME 中。因此，R1 的值必须从 EX/ME 前递到 EX 中 ALU 的两个输入端。在 C5 的开始，R1 的新值刚被存到 ME/WB。这个值可以通过图 3-5 中的 REG_data 进行前递。

现在我们再看 load 指令引起的依赖性，如表 3-7 所示，新值在 C4 结束时才可用，I3 和 I4 相对于 R1 的冲突可以借助于用于解决普通 ALU 指令依赖（寄存器 - 寄存器依赖）的前递技术来解决。

表 3-7 load 指令引起的寄存器 RAW 冲突

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1, 0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3, R1, #4		IF	ID	EX	ME	WB			
I3	LW R5, 0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6, R1, #20				IF	ID	EX	ME	WB	
I5	SUBI R1, R1, R7					IF	ID	EX	ME	WB

然而，我们仍然无法及时把 R1 的值前递给 I2。因为，I2 在 C4 开始时就需要值，此时正是 load 访问内存的时候。因此 I2 必须暂停一个周期，它对应的 EX 阶段将在 C5 开始，load 得到的值可以通过之前实现的硬件前递机制给它。在本例中，I2 在译码阶段多停留了一个周期，即在 C4 周期，I2 在 ID 阶段暂停。I3 则在 IF 阶段暂停，没有取新的指令。此时，I1 则通过 ME 阶段。指令的具体调度过程如表 3-8 所示。可以看到有一个周期被浪费了，因为没有取新的指令进来。

表 3-8 阻塞流水线来解决 load 带来的 RAW 冲突

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1, 0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3, R1, #4		IF	ID	ID	EX	ME	WB		
I3	LW R5, 0(R1)			IF	IF	ID	EX	ME	WB	
I4	ORI R6, R1, #20					IF	ID	EX	ME	WB
I5	SUBI R1, R1, R7						IF	ID	EX	ME

在图 3-6 中，我们为图 3-5 的基本五级流水线添加了前递和停顿的硬件支持。增加的部分做了重点标记，虚线给出了新增的控制线。

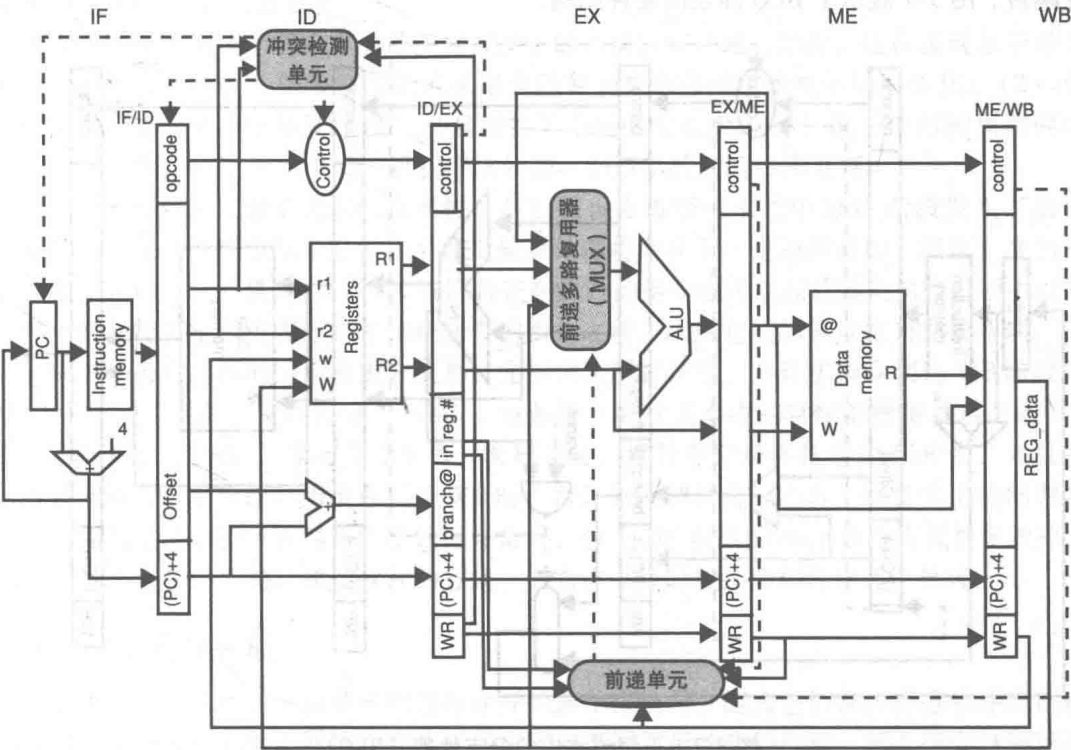


图 3-6 支持 load、store 和寄存器 - 寄存器指令的五级流水线结构示意图，含有避免 RAW 冲突的硬件支持

为了支持操作数前递，在 EX 中 ALU 的每个输入上都增加了一个三路复用器（为了清晰起见，图 3-6 中只画了前递 MUX）。每个多路复用器从下面几个输入中进行选择：（1）从 ID 段获得的 ID/EX 中的寄存器值；（2）锁存到 EX/ME 的 ALU 输出；（3）来自 WB 阶段的 REG_data。前递单元（FU）控制着这两个多路器。FU 将 EX/ME 以及 ME/WB 中的 WR 域和 EX 中指令的输入寄存器号进行比较，如果匹配，并且前递指令要往控制信号所表示的寄存器中写入值，那么 FU 就会设置多路器直接将值前递给 ALU 的输入。前递逻辑非常简单，其主要功能就是将操作数的值从流水线的的一个阶段传送到另一个阶段。

当冲突检测单元（Hazard Detection Unit, HDU）发现 Load 指令后紧跟着一条相关指令的话，就会暂停流水线。HDU 会检测 EX 阶段的指令是否是 Load 指令，以及其目的寄存器同 ID 中指令的输入寄存器是否相同。如果答案是肯定的，HDU 必须暂停 IF 和 ID 流水级，并向 EX 中插入一个 NOOP。为了暂停 IF 和 ID，HDU 暂时将 PC 和 IF/ID 的时钟无效掉。为了向 EX 中插入 NOOP（也被称作流水线气泡），HDU 还需要将 ID/EX 中的控制域设置为 0，以表明这条指令不会改变任何状态。注意流水线暂停甚至比前递的逻辑还简单，因为不需要通过总线发送任何操作数据。

控制冲突

到目前为止，我们都没有考虑跳转和分支指令的影响。分支指令尤其复杂，因为分支指令的执行需要确定分支条件和目标地址。分支指令的寄存器操作数的值可能依赖于之前的指令，由于数据可以被前递到 ALU 和 EX 的输入，因此，分支指令最早可以执行的时间点是 EX 阶段结束时。

分支指令的目标地址是在 ID 阶段计算出来的。分支默认是不跳转的，直到指令到达 EX 阶段，目标地址可以得到并且条件也由 ALU 计算完成时才能确定分支结果。以 BEQ 指令为例，两个输入寄存器的值在 ALU 中相减，ALU 输出中的 Z 位表明结果是否为 0，如果 Z 被置位，则分支跳转。图 3-7 展示了 BEQ 涉及的硬件结构。

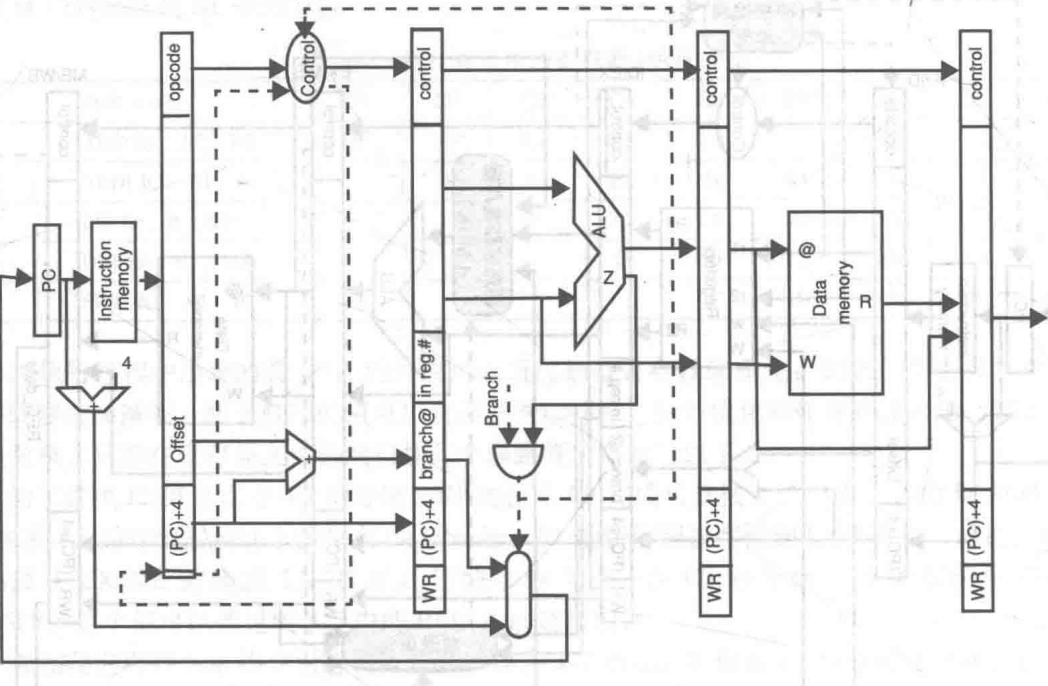


图 3-7 五级流水中的分支处理（BEQ）

在 EX 阶段要执行分支时, 必须进行以下两个操作。

- 目标地址必须在本周期结束时锁存到 PC。当 Z 的值为 1 并且 EX 中是分支指令时, 多路器将选择目标分支地址赋值给 PC。
- IF 和 ID 阶段必须清空。当 Z 的值为 1 并且 EX 中是 BEQ 指令时, 会往 ID 的 “control” 中发送一个取消 IF 和 ID 中当前指令的信号。无效掉 ID 中指令的机制同插入气泡一样 (将所有控制信号置为 0)。对于 IF 中的指令, 将在 IF/ID 中插入一个有效位, 并且在时钟周期结束时进行复位。在下一个时钟周期中, 当有效位被复位时, 会强行将 ID 中指令的控制位全部清零。

绝对地址的跳转指令可以在 IF 阶段即将结束时执行, 因为不需要计算地址。然而, 间接跳转必须在译码阶段结束才能执行, 因为必须先从包含目标地址的寄存器中取出跳转地址。在这种情况下, IF 中已经取到的指令必须被刷掉。

结构冲突

五级流水线不存在结构冲突。结构冲突是由对硬件资源的竞争产生的, 比如, 两条指令试图在同一拍使用同一资源。这种冲突可以通过延迟某一条指令的执行或者增加硬件资源来解决。

精确异常的处理

五级流水的精确异常可能会在 IF (缺页故障)、ID (未定义指令)、EX (算术溢出) 或者 ME (缺页故障) 中触发。精确异常不可能发生在 WB 阶段, 因为 WB 阶段唯一剩下的操作就是把结果写到寄存器中, 此时, 只有硬件错误才会触发异常。

当五级流水线中发生精确异常时, 硬件需执行以下操作:

- 出错指令以及按照进程序的所有后续指令都必须被清空 (清空其所在的流水级);
- 在出错指令之前 (按照程序序) 的所有指令必须执行完成;
- 开始执行异常处理程序。

非常有吸引力的一个解决方法是在异常发生的当前周期处理。然而, 这在实现起来是非常复杂的, 原因包括: (1) 需要清空的流水级会随着异常所在的流水级不同而变化; (2) 处理异常时必须获取异常条件和异常 PC, 并且应位于不同的流水级; (3) 同一个时钟周期内可能会出现多个异常; (4) 异常必须按进程序来处理, 而不是按时间序来处理。

为了深入理解上述最新需求的重要性, 我们假设在五级流水线中的 IF 阶段发生了缺页异常。如果异常在时钟周期结束时发生, 那么缺页处理将会在下一个周期开始。然而, 在当前触发例外指令之前、已经进入 ID 或 EX 阶段且正在执行的指令也可能在后续的执行过程中引发异常。由于这些指令按照进程序是在出现缺页的指令之前, 因此它们的异常应当先被处理。

解决上述各种问题的一个激进方法是, 先标识出所有异常, 并且在指令到达 WB 阶段之前保持 “沉默”。“沉默” 是指暂时 “忽略”。每条指令穿过流水线的时候都携带着自己的 PC 和异常状态寄存器 (ESR)。当指令发生第一次异常时, 将异常记录在指令的 ESR 中, 并且这条指令被替换成 NOOP 操作。当指令到达 WB 时, 再开始处理异常。还有一些技术上的问题需要注意, 比如当之前的指令在 WB 阶段处理异常时, 位于 ME 阶段的 store 指令应该被无效掉, 以避免其产生不应有的影响。通过这样的方式, 异常在 WB 阶段按照程序序逐个处理。

3.3.2 指令乱序完成

在五级流水中, 所有的指令按照进程序开始及结束执行, 因为它们都按照进程序依次通过流水线的每个阶段。如果不同指令在流水线中需要不同数量的时钟周期, 那么指令就很可能不再是按照进程序执行完成。图 3-8 展示的流水线例子中包括两条分别执行整型 (EX) 和浮点

(FP) 指令的流水线，其中一条用来处理 load、store、整数和分支指令，另一条用来执行浮点运算指令。根据指令的操作码，译码器决定每个周期是将指令送到整型流水线还是浮点流水线。处理器有两套分开的寄存器组：一组用于浮点操作数，一组用于整型操作数。所有指令会流经 ME 和 WB 这两个阶段。浮点指令的执行阶段需要 5 拍，并且全部流水化，因此浮点指令完成的时间可能比在它们后面进入流水线的整型指令（ALU、load/store）都要晚。所以，这种流水线和五级流水线的最大不同在于指令完成的顺序可能是乱序的。

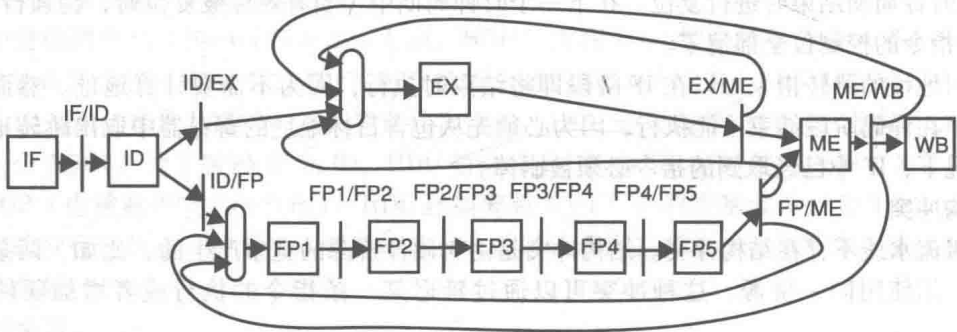


图 3-8 乱序完成的流水线

数据冲突

寄存器操作数的前递通路如图 3-8 所示。由 load 得到的操作数可以从 WB 前递到 FP1 或 EX（浮点或整型操作数）。整型和浮点结果也可以从输出前递到整型或浮点执行流水线的输入。由于浮点 store 的原因，前递实现逻辑比较复杂，因为浮点 store 在译码阶段同时需要整型寄存器（地址）和浮点寄存器（数据）的值。为了支持将数据前递给浮点 store，需要增加一条从 FP/ME 寄存器到 EX 输入端的通路。

如果发现和之前的且正在执行的指令存在寄存器相关（数据冲突）的话，ID 阶段的冲突检测单元（HDU）必须暂停本流水级的所有指令。对于寄存器来说，有两种可能的数据冲突：RAW 和 WAW。因此 ID 阶段指令的源寄存器和目的寄存器必须与正在执行的指令的目的寄存器进行对比检测，如果匹配则要暂停 ID 中的指令。指令的“操作延迟”这一概念是指，该指令在程序序中紧挨着的后续指令必须在 ID 中等待的时间，这个等待是为了避免其输入寄存器操作数存在 RAW 相关。在线性执行流水线中，如图 3-8 所示的流水线，指令的操作延迟（简称为延迟）是它的执行周期数减 1。

由于浮点指令的操作延迟较大（在本例中是 4 个周期），因此相比五级流水线来讲，寄存器 RAW 冲突会更频繁，如表 3-9 所示。

表 3-9 增加的 RAW 冲突停顿

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	L D F4, 0(R2)	IF	ID	EX	ME	WB						
I2	MULT. D F0, F4, F6		IF	ID	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	S. D F0, 0(R2)			IF	IF	ID	ID	ID	ID	ID	EX	ME

除了操作延迟，执行流水线的另一个重要特征是启动间隔，即发送两条同类指令到执行单元之间的最小间隔周期。在我们的简单机器中，启动间隔总是 1，因为整型和浮点单元为全流水执行，每个周期都可以发送一条指令。然而，如果某个执行单元不是流水化的，或者流水线不是线性的（复杂流水线含有反馈/前馈环，某些流水级会在多个不同周期中使用），启动间隔可能会比 1 大甚至是可变的（取决于之前发送的指令）。表 3-10 给出了我们的简单流水机器

上的操作延迟和启动间隔。由于所有的访存指令都按进程通过 ME 阶段，因此不存在内存的数据冲突。

表 3-10 操作延迟和启动间隔

功能单元	延迟	启动间隔
整型 ALU	0	1
load 单元	1	1
浮点操作单元 (OP)	4	1 (如果不支持流水的话, 则是 5)

结构冲突

整型和浮点寄存器组是分开的，因此即使每个寄存器文件只有一个写端口，在同一周期内，整数操作数和浮点操作数仍然可以同时更新寄存器文件。不过，浮点 load 或 store 可能会和之前的浮点运算指令同时到达 WB 阶段，从而导致对浮点寄存器文件写端口的结构冲突。表 3-11 中，I1 和 I5 在同一时刻 (C8) 到达 ME 阶段，但这不是问题，因为 I1 并不访存。然而，在下一拍，两条指令都执行写寄存器文件操作。如果它和之前发送的指令在 WB 阶段的寄存器写端口发生冲突的话，可以通过在 ID 阶段暂停一条指令的方法来避免。

表 3-11 寄存器堆写端口的结构冲突

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C8	C9
I1	ADD. D F1, F2, F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I2	ADD. D F4, F2, F3		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	L D F10			IF	ID	EX	ME	WB			
I4	L D F12				IF	ID	EX	ME	WB		
I5	L D F14					IF	ID	EX	ME	WB	

在 ME 阶段，浮点指令使用的总线不会访存，它和可能访存的整型总线是相互独立的。通过避免浮点寄存器文件写端口的冲突，就可以保证在 WB 阶段没有冲突发生。而在 WB 阶段，通过使用独立的整型和浮点总线，每个周期内就只有一条指令可以访问寄存器写端口，从而避免了冲突。

控制冲突

和五级流水线类似，控制相关的解决方案也是在 EX 阶段预测条件分支指令为 not_taken。如果分支结果是 taken 的，那么 EX 阶段会刷掉 IF 和 ID 阶段的指令，并执行跳转目标地址的指令。

精确异常

指令乱序完成的主要缺点是精确异常很难处理。因为指令是乱序通过 WB 阶段的，因此也无法像五级流水线那样等到 WB 阶段再处理。考虑下面的例子：

```
ADD.S F1,F2,F1
LW R2,0(R1)
```

这两条指令是完全独立的，可以被译码单元连续发射。由于 LW 在执行单元只花 1 个周期而 ADD. S 需要花 5 个周期，因此，LW 完成执行，通过 WB 阶段，以及流出流水线这几个步骤都发生在 ADD. S 完成执行之前（如表 3-12 所示）。如果 ADD. S 触发一个异常，并且暂时不处理直到 WB 阶段，那么 R2 的值会被 LW 更新并且整型寄存器的状态会在处理异常时被破坏。从图中可以看到 I2 在 C6 写结果，而此时 I1 还没执行完成。

表 3-12 精确异常的问题

时钟 ==>		C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD. S F1, F2, F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I2	LW R2, 0(R1)		IF	ID	EX	ME	WB			

在乱序完成的机器上实现精确异常太过复杂，以至于许多情况下直接不提供支持。硬件只是简单地告诉软件处理程序在某些 PC 附近发生了一个异常。当然，如果是这样的话，对于现在常见的由虚存（缺页）导致的异常，以及严格按照 IEEE 浮点标准的浮点指令导致的异常，是无法支持的。

另一种处理精确异常的方法是将它们看作流水线冲突。指令在 ID 阶段暂停，直到它之前所有的指令都没有异常为止才继续执行。要实现这种方法，所有潜在的异常都必须在指令执行期间尽早发现。很明显这种方法会降低流水线效率。

最后一种处理方式和五级流水线类似，那就是修改系统架构，使其在写回阶段按程序序遍历所有的异常。图 3-9 所示的流水线，将 EX 和 FP1 以及 ME 和 FP2 整合到一起，从而使得所有指令按序穿过所有流水阶段。这种方法需要更多的前递旁路，并且只适用于简单的流水线。另外，store 指令只能在其之前的所有指令都没有异常后才能发射。

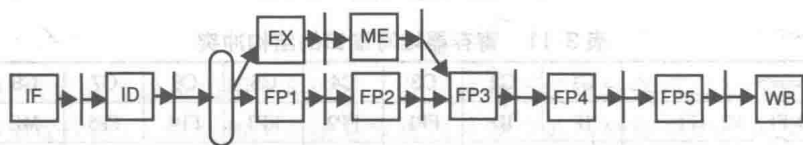


图 3-9 强制在 WB 阶段按序完成

3.3.3 超流水和超标量 CPU

在现有的技术下，五级流水的某些阶段可能会比其他阶段有更长的延时。例如，IF 阶段访问指令 cache 时间可能是译码或写回阶段的 2 倍。因为，在访问数据 cache 之前必须通过 TLB 进行虚实地址转换。当把五级流水中超过一个周期的阶段再流水化时，我们称这种 CPU 为超流水（superpipelined）CPU。超流水 CPU 的时钟比五级流水中原有最长延迟流水级的时钟要快。并且，通过将五级流水的每一级都再流水化，还有可能提高指令吞吐率。在执行时间的计算公式中，超流水技术会导致时钟周期变小，但是，CPI 却会增大。CPI 增大的主要原因是分支开销和操作延迟（按周期数算）的增加。在图 3-10 所示的超流水 CPU 中，分支（在 EX1 被执行）的开销扩大到了 3，寄存器-寄存器指令的操作延迟是 1，而 load 指令操作延迟是 3。



图 3-10 超流水 CPU

CPU 也可以是超标量的，这意味着在每个周期同时有多条指令被取指、译码和发射。这种 CPU 在执行既有整型操作也有浮点操作的负载任务时效率很高。图 3-11 展示了一个两路超标量流水线。在每个周期，可以取指和译码两条指令。如果第一条是整型或访存指令，第二条是浮点指令且和第一条相互独立，并且如果它们同流水线中其他指令没有冲突的话，那么就可以同时执行。

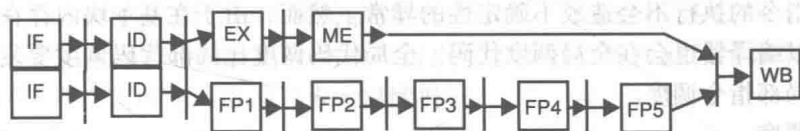


图 3-11 两路超标量 CPU

在超标量静态流水线中，对精确异常的支持更加复杂，设计实现超过两路的静态流水线也非常困难。实际上，整型流水线也可以被分成整型操作、访存操作和分支处理三条流水线，不过这种机器从来没有出现过。

静态超标量 CPU 可以在每个时钟周期执行多条指令，因此 CPI 一般都小于 1。相比于 CPI 而言，我们更多使用的是 IPC (Instruction Per Clock)，两者的关系为 $IPC = 1/CPI$ 。当然，超标量 CPU 同时也可以超流水的。

3.3.4 分支预测

本节描述的静态微结构的问题之一是对于分支指令每次都是预测为不跳转。分支预测逻辑在微架构中是硬化的，和编译器无关，因此，对于循环底部的条件分支，大部分情况下其实都是错误预测的，每次都导致两个周期的开销。硬化的分支预测是最静态的分支预测方案，因为它完全不受软件影响。

更灵活的硬件分支预测技术是基于操作码和/或基于分支指令的方向和地址偏移量大小。例如，BNEZ 指令可以预测为不执行跳转，而 BEQ 指令预测为执行跳转。同样，向后跳转的分支预测为跳转，向前跳转的分支预测为不跳转，偏移大的分支预测为不跳转。可以根据目标应用领域的典型基准测试程序的特征，在微结构设计时就确定好分支预测策略。

在编译时编译器控制分支预测也是有可能的，这种策略较为灵活且可以利用代码分析优势，编译器可以模拟特定代码特定输入时的执行，然后确定代码中每一个静态分支的最好预测结果，这一预测结果可以通过在分支指令中加入一个额外位（用于预测提示）的方式来传达给硬件。

图 3-5 ~ 图 3-11 中所示的结构中，分支预测效果非常有限，这主要是因为是在译码阶段就知道分支的目标地址，因此即使预测正确也只能节省一个周期。此外，一旦预测失败，则需要刷掉大量指令，这也使得该情况下的分支预测效果不大。

3.3.5 静态指令调度

图 3-8 ~ 图 3-11 的流水线中，指令完成译码后，假如它们之间不存在相关，就开始以程序顺序依次执行。

在静态指令调度方法中，硬件没有动态识别代码执行顺序的机制。指令是以编译器的生成顺序调度执行的。为了尽量减少译码阶段由于指令相关导致的阻塞，需要依赖编译器来做指令执行的调度。这就是为什么将其称之为静态调度流水线或静态流水线。

编译器可以在局部（在基本块内）或者全局（在基本块间）范围调度代码。基本块是静态代码中的一系列连续指令，如果基本块内的某条指令执行了，那么其他指令也会一起执行。所以基本块（除了在其最底部的指令外）不能包含分支跳转指令。并且，在基本块内（除了在其最顶部的指令外）不存在分支跳转的目标指令。不幸的是，基本块的平均大小比较小，动态指令混合表明指令中有 20% ~ 25% 是分支指令。所以基本块的平均大小最多才包含 5 条指令。在基本块内的静态指令调度称为局部调度。编译器在基本块内移动指令是安全的，因为当

代码运行时，指令的执行不会造成不确定性的异常。然而，由于在基本块内存在很多指令级并行的限制，所以编译器也会在全局调度代码。全局代码调度比局部代码调度要复杂得多。下面我们先看一下局部指令调度。

局部指令调度

举一个简单的例子，这个例子实现内存中两个向量相加：

```
for (i=0;i<100;i++)
    A[i] = A[i] + B[i];
```

一个完全没有优化的简单汇编代码实现这个循环，如下所示：

```
Loop    L.S F0,0(R1)           (1)
        L.S F1,0(R2)           (1)
        ADD.S F2,F1,F0         (2)
        S.S F2,0(R1)           (5)
        ADDI R1,R1,#4          (1)
        ADDI R2,R2,#4          (1)
        SUBI R3,R3,#1          (1)
        BNEZ R3,Loop           (3)
```

在括号里，我们标明了在单发射流水线机器上指令执行需要的时钟周期数。对于非分支指令，这里的时钟周期就是它在译码阶段所花费的时间，包括由于和之前指令有数据冲突造成的阻塞时间。对于分支指令，可能是 1 个时钟周期（分支未跳转）或者是 3 个时钟周期（分支跳转），这是因为在取指和译码阶段中的两条指令在每次分支跳转之后都要被刷掉。上述循环的每次迭代所用的时钟周期数是 15，所以 CPI 是 $15/8 = 1.875$ 。

由于分支每次都是预测不跳转，所以大多数情况下分支指令之后的两条指令都被分支给刷掉了。解决办法之一是重构编译后的指令，把分支指令放到循环体的最上面，这样在大多数情况下分支都是不跳转的。

一些指令集采用延迟分支技术，该技术会延迟分支指令的影响。比如，分支被延迟一条指令的意思是，在分支指令后的一条指令总是执行，然后分支才被处理。假如分支被延迟两条指令，编译器就应当总能将两条指令移到延迟槽中，这样的话，条件分支的代价就总是只有一个时钟周期。然而，编译器也不是总能找到有用的指令来填充延迟槽，此时编译器就必须在延迟槽中填充无操作指令（NOOP），其效果等价于刷掉流水线中的两个流水级。除此之外，延迟分支还有其他的一些问题（比如异常处理）。

不过至少编译器可以在循环体内对代码进行移动，比如用一些非常简单的代码移动和转换技术。在这个例子中，编译器可以首先考虑将 SUBI 指令移到上面去，因为没有相关冲突，编译器可以一直把它移动到第 2 个 load 之后。类似地，两条 ADDI 指令可以移动到 store 指令之前。不过这里有个小问题需要注意：在 R1 寄存器上会存在 WAR 相关，因为更新 R1 寄存器的 ADDI 指令被移到了 store 指令之前，而 store 指令也需要用到 R1 寄存器的内容。这个相关可以通过调整 store 指令的偏移量来轻松解决，调整后的代码如下：

```
Loop    L.S F0,0(R1)           (1)
        L.S F1,0(R2)           (1)
        SUBI R3,R3,1           (1)
        ADD.S F2,F1,F0         (1)
        ADDI R1,R1,#4          (1)
        ADDI R2,R2,#4          (1)
        S.S F2,-4(R1)          (3)
        BNEZ R3,Loop           (3)
```

如上所示，代码中每次迭代所用的时钟周期变成了 12，所以 CPI 是 $12/8 = 1.5$ ，速度提升了 25%。这一提升幅度不容小视，而且我们只付出了很少的代价。假如编译器可以越过基本块边界移动代码，那么速度就可以进一步大幅度提升。

全局指令调度

全局指令调度比局部指令调度更强大，因为相对于在基本块内的编译优化方法，全局指令调度可以调度更多的指令。静态调度的一个主要优化目标是循环体，循环体可以在源代码级进行识别，所以可能用于编译器优化。针对循环体的全局调度也称作循环调度。循环展开和软件流水就是两类著名的循环调度技术。而非循环调度技术则包含踪迹调度。

在循环展开技术中，循环体被展开若干次。为了避免一些本不需要展开的例外情况，原循环展开的次数必须恪守某一精确值——既不能多，也不能少。

重新回顾上面实现两个向量相加的代码。在图 3-12 中，我们将循环体展开了两次。在第一次（图 3-12a）编译器对代码进行了复制，同时移除了分支和地址寄存器的冗余更新，调整了两个 store 的偏移量。基本块的大小现在也几乎增加了一倍。然而，由于寄存器上的 WAW 和 WAR 相关，代码在新的更大的基本块上也很难再移动了，比如编译器不能将第三条 load 指令移到第一条 ADD.S 上面，因为在 F0 上存在着 WAR 相关。同样第一条 store 指令也不能移到第二条 ADD.S 的下面，因为在 F2 上存在着 WAW 相关。

循环展开两次	重命名浮点寄存器	重新调度
L.S F0,0(R1)	L.S F0,0(R1)	L.S F0,0(R1) (1) (1)
L.S F1,0(R2)	L.S F1,0(R2)	L.S F1,0(R2) (1) (1)
ADD.S F2,F1,F0	ADD.S F2,F1,F0	L.S F3,#4(R1) (1) (1)
S.S F2,0(R1)	S.S F2,0(R1)	L.S F4,#4(R2) (1) (1)
L.S F0,#4(R1)	L.S F3,#4(R1)	ADD.S F2,F1,F0 (1) (2)
L.S F1,#4(R2)	L.S F4,#4(R2)	ADD.S F5,F3,F4 (1) (2)
ADD.S F2,F1,F0	ADD.S F5,F3,F4	SUBI R3,R3,#2 (1) (1)
S.S F2,#4(R1)	S.S F5,#4(R1)	ADDI R1,R1,#8 (1) (1)
ADDI R1,R1,#8	ADDI R1,R1,#8	ADDI R2,R2,#8 (1) (1)
ADDI R2,R2,#8	ADDI R2,R2,#8	S.S F2,#-8(R1) (1) (1)
SUBI R3,R3,#2	SUBI R3,R3,#2	S.S F5,#-4(R1) (1) (1)
BNEZ R3,Loop	BNEZ R3,Loop	BNEZ R3,Loop (3) (4)
拷贝循环体两次， 删除冗余指令， 调整跳转偏移	消除WAW和WAR相关	load上移，store下移， 调整跳转偏移
a)	b)	c)

图 3-12 循环展开示例

为了消除这些假相关（WAR 和 WAW），编译器对浮点寄存器进行了重命名，如图 3-12b 所示。重命名是避免假相关（名字相关）问题的通用技术。假相关及其造成的冲突是由于存储限制（寄存器或内存）引起的。如果有无限多的寄存器或内存，那么编译器就总能将新值分配到新的寄存器或者内存地址，所以 WAR 和 WAW 相关就能消除。在图 3-12 中，新的浮点寄存器被分配给循环体的第二次迭代，因此在新的循环体中寄存器 F0、F1 和 F2 被重命名为 F3、F4 和 F5。现在 load 可以被移上去了，而 store 可以被移下来，如图 3-12c 所示。这个代码在图 3-8 中的 CPU 流水线中不会阻塞。需注意的是，store 指令的地址被重新调整了。在每个指令后面的括号里显示了在译码阶段每个指令花费的时钟周期数（第一个数字），展开后循环体迭代一次需要 14 个时钟周期，相当于 7 个时钟周期执行一次原先循环的迭代。和没有优化的循环体相比，加速比是 $15/7 = 2.14$ 。

图 3-12 的第二个括号的数字显示的是图 3-10 超流水线中每个指令需要的时钟周期数，假

定浮点操作还是五级流水。在超流水线中执行展开后循环体的一次迭代需要 17 个时钟周期，相当于 8.5 个时钟周期执行一次原先循环的迭代。但因为时钟频率是原来的 2 倍，所以加速比是 $2 \times 15 / 8.5 = 3.25$ 。

代码在图 3-11 的超标量机器上表现得不是很好，这主要是因为代码中浮点指令的比重很小。图 3-13 给出了调度过程以及真实程序。可以看到展开后的循环体每次迭代需要 14 个时钟周期，所以加速比是 $30 / 14 = 2.14$ 。

调度过程		真实程序
L.S F0,0(R1)	(1)	L.S F0,0(R1)
L.S F1,0(R2)	(1)	L.S F1,0(R2)
L.S F3,#4(R1)	(1)	L.S F3,#4(R1)
L.S F4,#4(R2)	(1)	L.S F4,#4(R2)
SUBI R3,R3,#2	(1)	SUBI R3,R3,#2
ADDI R1,R1,#8	(1)	ADD.S F2,F1,F0
S.S F2,#-8(R1)	(1)	ADDI R1,R1,#8
S.S F5,#-4(R1)	(3)	ADD.S F5,F3,F4
BNEZ R3,Loop	(3)	ADDI R2,R2,#8
		S.S F2,#-8(R1)
		S.S F5,#-4(R1)
		BNEZ R3,Loop
a)		b)

图 3-13 在双发射超标量机器中的循环展开

当迭代之间相互独立的时候，循环展开比较合适，此时没有跨迭代之间的依赖。跨迭代依赖是指在循环中不同迭代的语句之间的相关性（相对于循环内相关而言，后者也是之所要做循环展开的原因）。举个例子，一个简单的循环如下：

```
for (i=5;i<100;i++)
    A[i] = A[i-5] + B[i];
```

假如这个循环展开 5 次，第五次迭代将对第一次迭代的 RAW 相关，因此限制了指令级并行的程度。第五次循环迭代必须等待第一次循环迭代的结果，因此循环中的跨迭代依赖限制了循环展开的效果。

循环展开的另一个限制是 ISA 中可寻址的寄存器数量，因为在移动代码时，不可避免地要做寄存器重命名。还有一个缺点是循环展开会导致代码量的膨胀。

另一个循环调度技术是软件流水，这种技术将原有循环转换成另一个循环，新的循环将原先循环中具有相关性的指令放在多个迭代间流水执行。比如上一个静态代码调度的例子，编译器可以将循环变成两个部分：第一个部分由两个 load 和一个 Add 组成，第二个部分由一个 store 组成。

表 3-13 中每一列对应于原循环的每一次迭代（O_ITE），而每一行对应流水循环的每一次迭代（P_ITE）。为了正确开始和结束流水循环，前后还各需要增加一个前序（prolog）和收尾（epilog）的代码片段。请注意，不管这份代码是按列执行还是按行执行，执行的都是相同的代码序列。流水循环代码如下：

```
Prolog: L.S F0,0(R1)
        L.S F1,0(R2)
        SUBI R3,R3,1
        ADD.S F2,F1,F0
        ADDI R1,R1,#4
        ADDI R2,R2,#4
```

```
Loop      S.S F2,#-4 (R1)      (1)
          L.S F0,0(R1)         (1)
          L.S F1,0(R2)         (1)
          SUBI R3,R3,1         (1)
          ADD.S F2,F1,F0       (1)
          ADDI R1,R1,#4        (1)
          ADDI R2,R2,#4        (1)
          BNEZ R3,Loop         (3)

Epilog:   S.S F2,#-4(R1)
```

新代码中的主循环避免了原循环中的阻塞，并且没有使用额外的浮点寄存器。每个迭代的执行时间是 10 个时钟周期，相对于原程序加速了 50%。和循环展开技术不同，软件流水生成的新循环代码其大小和原循环是相同的，对地址寄存器和分支的操作次数也都没有改变。不过，我们需要两个额外的代码片段（称为 prolog 和 epilog）来启动和结束软件流水。

表 3-13 软件流水示例

	O_ITE1	O_ITE2	O_ITE3	O_IT4
Prolog	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	—	—	—
P_ITE1	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	—	—
P_ITE2	—	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	—
P_ITE3	—	—	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0
Epilog				S.S F2,0(R1)

为了进一步提升指令级并行度，大幅提高超流水/超标量处理器的性能，编译器可以同时采用循环展开和软件流水技术。不过，采用的顺序很重要——首先是循环展开，然后是软件流水。这是因为软件流水会引起跨迭代的依赖，从而降低循环展开的效率。

3.3.6 静态流水线的优缺点

静态流水线最主要的优点是硬件简化，因此相对于复杂硬件来说具有时钟频率上的优势。因为静态流水线简单，其性能也是可预测的，而且编译器可以利用代码的全局信息来静态地优化性能。不仅如此，静态流水线消耗的能量/功耗也更少，因为很多在复杂硬件上动态执行的活动可以转移到编译器上完成。

不幸的是，静态调度流水线并不擅长处理动态事件，比如条件分支，异常和 cache 失效等。事实上，本部分讨论的微架构机制都不能处理 cache 失效（包括 I-cache 和 D-cache miss）。这里解决 cache 失效的唯一方法是冻结处理器，重填 cache，然后再重新执行这个周期的操作。静态流水线一次也只能访问一次主存，因此无法有效解决内存墙问题，除非在核内支持多线程技

术，在一些现代处理器中应用了这项技术，我们将在第 8 章中进行介绍。

静态（编译器）指令调度还受限于编译时动态信息的缺失，比如内存地址。当访存指令在代码中需要被移动时，需要内存地址来检测并解决访存操作的相关性。

静态流水线所需的编译器也会更加复杂，尤其是在代码的生成阶段和优化阶段，后者非常依赖于微架构。利用编译器来优化性能这是可以接受的，然而，争议在于是否需要编译器来确保程序执行的正确性。比如，一些静态机器可能在硬件上不支持前递或流水线阻塞，而是依赖编译器在两条相关指令之间插入不相关指令或者 NOOP 指令来确保其满足指令延迟的需要。这个方法虽然简化了硬件，但是编译生成的二进制文件在指令集相同但指令延迟不同的机器之间却不可移植，这违背了向后兼容的原则。这种情况下，当把二进制文件从某一机器上移植到另一机器上时，二进制文件必须进行静态翻译或者动态解释执行。

最后，也许是最重要的，假如指令不是按照程序序完成执行的，精确中断就难以有效处理。随着流水线深度的增加以及指令发射宽度的提高，复杂性也会增加。鉴于此种原因，由于硬件的简单性和功耗优势，静态流水线普遍应用于嵌入式领域，而动态流水线则更多地用于通用环境。当然，由于核内多线程技术的出现，这种趋势也可能在将来发生逆转。

3.4 动态调度流水线

在静态架构中，指令需要等到与之前发射的指令没有相关性时才能发射。结构相关、数据相关和控制相关在译码阶段会全部解决。一旦指令通过译码阶段就不会阻塞了，除非发生了异常。因此，静态流水线只能通过发掘基本块内的指令级并行（ILP）来实现性能优化，当然基本块可以通过编译器来扩展，尤其对于循环体而言。

为了提高指令级并行度，指令应该可以在满足相关性的前提下以任何顺序执行，而不是一定要按照程序序执行。图 3-14 展示了动态调度指令是如何提高图 3-8 中简单流水线的执行效率的。这是一个将两个向量相加的简单循环程序，图 3-14 中标明了在静态流水线中，译码阶段每条指令需要的时钟周期数。由于与加法指令的相关性，store 需要 5 个时钟周期。正如图中显示的那样，虽然 store 在译码阶段被阻塞了，但是后续的指令还是能够被取到，并且绕过 store 指令。这两条 add 指令，一条 sub 指令，甚至还有一条分支指令（需要小心），都可以在 store 指令发射之前就进行译码和发射。这样的话可以将执行周期数压缩 4 个变成 11 个时钟周期，从而获得比局部静态调度更好的性能。

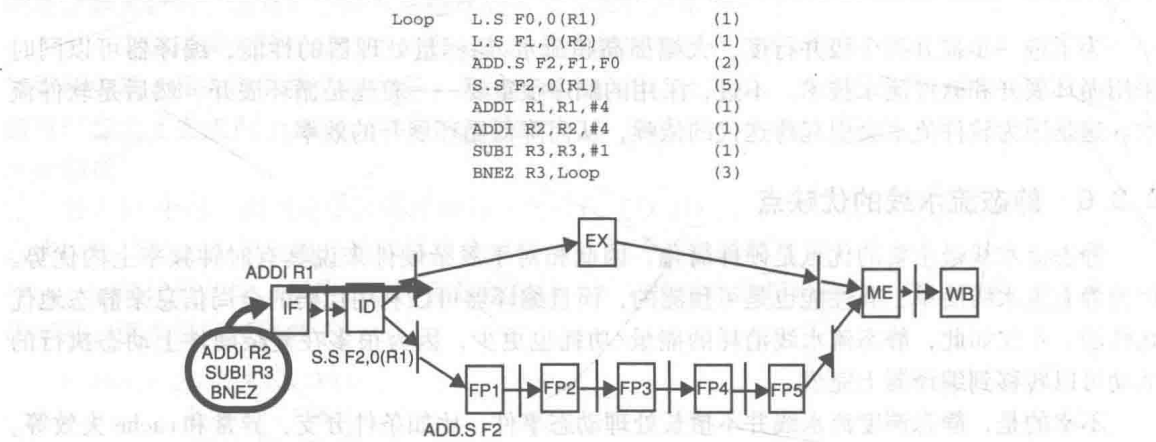


图 3-14 乱序执行

不过这里还有一个小问题，那就是 R1 上 store 和 add 指令 WAR 相关。编译器能够调整

store 的偏移量，但是静态硬件通常无法动态完成这些。不仅如此，当越来越多的指令被阻塞并且能够通过译码阶段的旁路发射出去时，硬件的复杂度也会迅速提高。因此，一个新的、可扩展的动态调度方法是必需的，整个流水线架构需要从头考虑。

对于这一新的流水线架构有几个基本的需求。为了支持乱序（OoO）执行指令，有些问题必须解决：第一，所有的数据相关需要解决。RAW、WAW 和 WAR 数据相关无处不在，它们存在于寄存器和内存操作数中。第二，控制相关一定要解决。指令一定要能够跨越条件分支执行，否则，指令级并行会被基本块的大小所限制。第三，结构相关必须解决。两个指令不能在相同的时钟周期占用相同的硬件资源。第四，对于某些异常来讲，必须确保精确异常模型。显而易见，图 3-14 中的简单流水线结构没有解决这些问题的资源，我们需要一个完全不同的硬件架构来解决上述问题。

3.4.1 解决数据相关：Tomasulo 算法

图 3-15 中的微架构受到了 Tomasulo 算法的启发，该算法最早是在 20 世纪 70 年代部署在 IBM3033 模型上的。它解决了寄存器和访存操作数因为乱序执行产生的数据相关，但它没有解决由于条件分支或异常引起的控制相关。

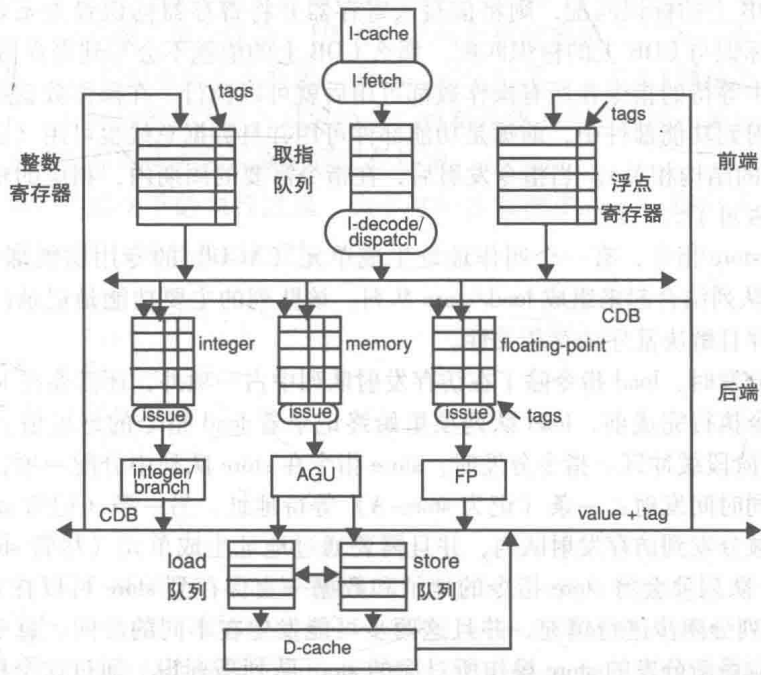


图 3-15 Tomasulo 算法的硬件结构

在机器流水线前端（front-end）中，指令顺序从指令 cache 中获取，然后被放置到一个先进先出（FIFO）的取指队列（IFQ）里。队列前面的指令会被译码然后分发（dispatch）给三个发射队列中的一个（取决于操作码）：一个针对整型和分支指令的发射队列，一个访存指令的发射队列，以及一个浮点指令发射队列。指令在相应的发射队列中进行等待，直到其源操作数可用。源操作数包含寄存器 - 寄存器指令和分支指令的源寄存器、load 指令的地址以及 store 指令的地址和待写入值。指令执行的结果通过公用数据总线（CDB）传输给分散在各个队列中等待的指令。一旦所需的操作数全部就绪，指令就可以在队列对应的功能单元中执行了。当执行完成后，指令通过公用数据总线将写寄存器的结果写入所有需要该结果的发射队列项中，这

个寄存器写操作发生在时钟周期的后沿。

分发逻辑在将指令送给机器流水线后端之前可以发现所有的寄存器相关。机器的后端主要由发射队列和功能部件组成。每个发射队列项关联一个 ID（在本例中 ID 至少需要 4 位，以区分队列中的总共 15 项）。指令从分发时就先保留一个发射队列项，到执行结束后再释放，并且指令的目的寄存器在其分发时由对应的队列项 ID 来标识。这些标识也同寄存器堆中的寄存器相关联。

分发逻辑首先根据指令操作码分配一个发射队列项。之后，它将这条指令的源寄存器连同其标识取出。如果输入寄存器的标识为有效，那么这个寄存器的值是一个旧值，并且新值在后端待定。这种情况下，分发逻辑将这个标识保存在发射队列项的输入操作数域中，并且设置该项为未就绪。否则，若寄存器的标签是无效的，那么这个寄存器中的值是正确的，分发逻辑就会将寄存器值存放在调度队列项的操作数域，并标为就绪。最后，分发逻辑将发射队列项的 ID 保存在指令目的寄存器的标识域中。

当指令的写寄存器结果在公用数据总线上传输时，其值和目的寄存器标识都会被放到总线上。总线上的标识值和所有发射队列中的所有操作数项进行匹配，并将值写到所有匹配成功的项中，同时将其标记为就绪。另外，寄存器堆也通过总线上的标识对寄存器进行访问。如果寄存器的标识同 CDB 上的标识匹配，则将值写入寄存器并将寄存器标识置为无效。否则，如果寄存器组中没有标识与 CDB 上的标识匹配，那么 CDB 上的值就不会写到寄存器中。

在发射队列中等待的指令在所有操作数都可用后就可以执行。在操作数就绪的下一拍，指令可以被选中发射到功能部件中，前提是功能部件可用并且数据总线也可用（上述条件消除了功能部件和 CDB 的结构相关）。当指令发射后，在指令需要的周期内，相应的功能部件和公用数据总线就都被占用了。

对于 load 和 store 指令，有一个叫作地址生成单元（AGU）的专用功能部件来生成地址。load 队列和 store 队列结合起来组成 load/store 队列。该队列的主要功能是记录已发射的和待发射的内存访问，并且解决乱序访存相关性。

在进行指令分发时，load 指令除了在访存发射队列中占一项外，还需要在 load 队列中分配一项。一直到指令执行完成前，load 队列项里始终记录着 load 指令的地址值。它也可以作为 cache 返回数据的阶段缓冲区。指令分发时，store 指令在 store 队列中分配一项，并且被拆分成两条子指令在不同时间发射：一条（记为 store-A）等待地址，另一条（记为 store-D）等待数据。两条指令都被分发到访存发射队列，并且需要通过地址生成单元（尽管 store-D 指令绕过了加法器）。store 队列项会将 store 指令的地址和数据一直保存到 store 可以在 cache 中执行为止。因此 store 队列分两步进行填充，并且这两步可能发生在不同的时间。每一条分发的 load 指令也会携带对应最新分发的 store 操作所对应的 store 队列项标识。通过这个标识，load 指令可以搜索到所有先于它执行但还没有写到 cache 的 store 指令。

访存操作数引起的 RAW、WAW 和 WAR 相关可以通过 load/store 队列发现和解决。准备好的 load 可以发射到 cache 中，当且仅当队列中在它前面没有相同地址的 store 指令；准备好的 store 指令可以发射到 cache，当且仅当队列中在其之前没有相同地址的 load 或 store 指令。实际中，同一时间相同地址的 load 和 store 都在等待执行的情况很少见。然而，主要问题是队列中某些 load 或 store 的地址可能是未知的，比如还没计算出来。在此情况下，为了安全起见，硬件必须假设这些地址一样。这种策略十分保守并且限制了 load 和 store 在 cache 中的乱序执行。上面这种通过比较访存地址来避免相关性的过程叫作内存歧义消除。

最后，分支指令被当作整型指令对待。它们在发射队列中等待操作数，然后，通过公用数据总线将结果传送给分发逻辑。分发逻辑在处理分支指令时阻塞，并且等待其结果。如果分支

没有跳转，分发逻辑仍然从队列中继续调度指令。如果分支出现跳转，分发逻辑将清空取指队列，指导取指单元从目标地址取指并且填充取指队列。

Tomasulo 算法通过寄存器重命名来消除寄存器操作数的相关性。在任意时刻，在后端可能有多对对应同一寄存器的结果，分发逻辑可以将寄存器重命名到某个产生该值的发射队列项。

例 3.1 Tomasulo 算法中的寄存器重命名 在下面的程序中，我们解释了寄存器值相关 (RAW, WAW, WAR) 是怎样通过 Tomasulo 算法一步一步解决的。

```
L.S F0,0(R1)
ADD.S F1,F1,F0
L.S F0,0(R2)
```

按照这个序列，第二条 load 指令将数据存储到 F0 可能只要几个周期 (D-cache 中命中)，而第一条 load 的 D-cache miss，可能需要数百个周期。所以第二条 load 指令在第一条 load 指令完成之前早就执行完了。然而，ADD.S 依然可以读到正确的值，因为它会通过公用数据总线上相应的标识值等待第一条 load 得到的结果，并且第二条 load 对应不同的发射队列项，因此有不同的标识。这将确保第一条 load 指令和 ADD.S 在 F0 上存在 RAW 相关，同时也会避免 ADD.S 和第二条 load 之间在 F0 上存在 WAR 相关。此外，最终 F0 的值将会是第二个 load 产生的值，因为在第二个 load 指令被分发后，F0 的标识将会被改成第二个 load 指令在发射队列项中的标识，并且 F0 等待数据总线上出现第二个 load 产生的值。一旦 F0 通过数据总线得到第二条 load 指令的值，它的标识将会无效并且忽略之后数据总线上出现的第一条 load 产生的值，这样第一条 load 指令产生的值就不会被存储到寄存器中。

例 3.2 Tomasulo 算法下的执行过程 为了进一步掌握 Tomasulo 算法的工作过程，我们将一步一步跟踪以下代码片段的执行过程：

```
Loop L.S F0,0(R1)
      L.S F1,0(R2)
      ADD.S F2,F1,F0
      S.S F2,0(R1)
      ADDI R1,R1,#4
      ADDI R2,R2,#4
      SUBI R3,R3,#1
      BNEZ R3,Loop
```

整型和分支指令执行时间是 1 个时钟周期。浮点指令执行过程流水化且需要 5 个周期。所有的 cache 访问 (指令和数据) 命中时间是 1 个周期。store 指令被分成两条子指令：一个计算地址，一个获取数据，两者都会被分发到访存队列。AGU 和 cache 阶段分别需要 1 个周期。分发 (1 个周期) 是一项复杂操作：它要取寄存器值，分配发射队列和 load/store 队列资源，并且重命名寄存器。当发射队列 (对所有指令来说) 或 load/store 队列 (只针对访存指令来说) 满时，会暂停从取指队列中进行指令分发。发射 (或调度) 也需要 1 个周期，包括从发射队列选取就绪指令，解决功能部件和公用数据总线的冲突，以及分配一条选中指令到空闲的功能部件。指令执行的结果在数据总线上的传输也需要一个周期。

表 3-14 给出了每条指令执行的不同阶段的周期数。指令按照处理程序序或者分发顺序列在连续行中。每一列对应一个流水级，从分发到写结果：每个表项给出了指令流过该流水线级时的时钟周期数，周期 1 开始执行第一个 load 的分发。地址生成单元可以认为是 load 和 store 的执行单元。当条件满足时，紧接着开始 cache 访问。该表按时钟周期逐步填充，每次发射一条指令时，必须确保预留其所需的资源 (比如第一级的执行单元、cache 以及公用数据总线

等)。这些需预留的资源在括号中做了标记。

表 3-14 Tomasulo 算法执行示例

		分发	发射	开始执行	完成执行	cache	CDB	备注
I1	L S F0, 0(R1)	1	2	(3)	3	(4)	(5)	
I2	L S F1, 0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD. S F2, F1, F0	3	7	(8)	12	—	(13)	等待 F1
I4	S. S-A F2, 0(R1)	4	5	(6)	6	—	—	
I5	S. S-D F2, 0(R1)	5	14	(15)	15	(16)	—	等待 F2
I6	ADDI R1, R1, #4	6	7	(8)	8	—	(9)	
I7	ADDI R2, R2, #4	7	8	(9)	9	—	(10)	
I8	SUBI R3, R3, #1	8	9	(10)	10	—	(11)	
I9	BNEZ RS, Loop	9	12	(13)	13	—	(14)	等待 R3
I10	L S F0, 0(R1)	15	16	(17)	17	(18)	(19)	等待 I9 (在分发级)
I11	L S F1, 0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD. S F2, F1, F0	17	21	(22)	26	—	(27)	等待 F1

从这个表中我们可以得出关于这种体系结构行为的一些结论。最重要的是，由于管理指令执行的开销，操作延迟被拉长了。例如，即使寄存器 - 寄存器指令只要 1 个周期并且在静态流水中的操作延迟可以是 0，然而在这里，有相关的指令必须在执行之前等待 3 个周期，因此有效的操作延迟是 2 个周期。Tomasulo 算法的优点是可以乱序执行，这在表中的“开始执行”这一列就可以明显看出来。这次计算中的关键路径包括 load、之后的 ADD. S，以及再之后的 store，其他所有指令可以和该关键路径并行执行。

分支指令起到了类似“屏障”的作用，指令级并行很难跨越基本块进行挖掘。为了将分支指令的影响降到最低，当分发逻辑在等待分支结果的时候，取指阶段可以同时为目标地址（分支跳转）和连续地址（分支不跳转）两个方向的指令进行预取和译码，这样一旦分支结果出来之后，无论跳转与否，待执行的指令都已经完成了译码阶段。

最后，这个动态微架构也可以利用静态（编译器）调度的结果。比如，如果编译器将判断分支是否跳转的减法指令提前，那么分支指令的结果提前几个周期便可得知。

3.4.2 推测执行

典型的基本块都太小了，以至于无法提供足够的机会来完全利用动态调度处理器的优势。然而要想在分支指令结果出来之前还能继续执行的话，还面临很多挑战。在分支结果确定之前就继续执行也称为推测执行（speculative）或“猜测执行”。“推测指令执行”的语义表明这个执行可能是无效的，因为指令该不该执行在执行的时候还是不确定的。

一旦指令被推测执行了，那么在这条指令确定应该被执行之前，微结构都必须有办法消除推测执行指令的影响。指令执行的影响主要包括对存储的修改，主要是内存地址和指令集可见的寄存器和异常等。在确定某个修改能回滚之前，推测执行指令是不能修改存储器的，由推测执行指令引发的异常也必须丢弃。通常做法是，为推测指令执行的结果提供一个临时存储位置来保护存储器，当推测指令成为确定指令的时候，再把这些临时结果提交到结构状态中。除了临时存储，该策略还需要一个灵活有效的技术来对推测指令进行回滚。

推测执行的方法可能有很多种。在程序执行的任何时刻来看，程序的未来执行过程都可以看成是一棵由基本块组成的树，如图 3-16 所示。树里面的每条边代表了一个基本块，每个节

点代表了一个二路分支。在树的最顶端是当前块和当前 PC 指针，这个指针指向基本块的第一条指令。在当前基本块内执行的指令是非推测的，但是下面的指令执行都是推测的。

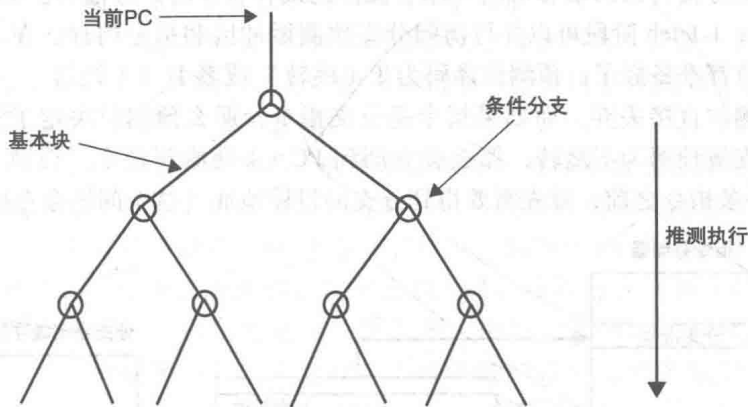


图 3-16 推测执行表示成基本块构成的树

推测执行的可能方法之一是全路径执行，在这种方法中，每个分支的两边都被执行到，从而遍历了整棵树。随着分支条件结果的确定，一些执行过的推测指令变成非推测的，它们的结果会被提交，而其他的则被丢弃。虽然全路径执行方法有可能达到最高的性能，但是非常复杂，实现代价也很高。之所以复杂是因为微架构必须跟踪基本块在树中的执行轨迹，使它能够划分哪些存储更新需要提交，哪些异常是有效的。不过全路径执行的最大问题还是高昂的代价。这是因为虽然有用的指令数目随着树的深度线性增长（某条路径上的指令数），但同时推测指令数却随着树的深度呈指数增长（一棵树中指令总数）。潜在的速度提升随着树的深度（或者推测执行的层数）增加而很快饱和，因为与此同时所需的功能部件以及控制和取消部件的数量会呈现爆炸性增长。所以往前推测得越远，执行的有效性就越是急剧降低。

一个更经济的方法是在执行树中只跟踪最有可能执行的路径——单路径。这种方法通过预测条件分支的输出来实现，具体的做法是每当条件分支取指和译码之后，就对其结果进行预测，并沿着预测路径按照贪心算法不断往下执行，并逐个处理碰到的分支指令。通过这种方法，推测执行指令的数目会和推测执行的层数成正比，这样就比全路径有效多了。不仅如此，由于推测执行的指令序列是线性的，所以跟踪、回滚都很容易，而且控制异常也简单多了。这种单路径的方法被大多数的商业微架构所采用。

还有一种折中的方法是跟踪部分路径而不是所有路径。大部分分支可以很准确地进行预测（不管是跳转还是不跳转），在这种情况下，就只处理最有可能的路径。然而，仍然有少数分支是无法预测的。目前，不管是静态分支预测算法还是动态分支预测算法都已经变得非常复杂了，进一步的性能提升只能期望于解决不可预测（也即执行概率各半）的分支。解决这种分支的方法之一是对分支的两侧都推测执行，当分支条件明朗以后再丢弃不需要执行的一侧，这被称作多路径执行算法。不过这种方法仍然比较复杂，主要是因为跟踪和回滚非线性增长的推测指令序列本身就很复杂。

3.4.3 动态分支预测

对于任何带有推测执行功能的高效微架构而言，好的分支预测算法都是一个至关重要的因素。分支预测可以是静态或者动态的，静态预测可以是硬化的或者基于编译器的。在 3.3.4 节中，我们看到一些基于指令混合和指令剖析的静态技术。静态技术无法适应动态情况。对于执

行概率各半的分支，静态预测难以处理，而对于动态预测来讲可能很容易。比如，对于一个执行概率各半的分支，可能执行程序的前一半部分都会跳转，而后一部分都不跳转，这种动态行为只有通过能跟踪分支行为的动态机制（基于硬件的或者软件的）才能捕获到。

在图 3-17 中，I-fetch 阶段可以并行访问分支预测缓冲区和指令内存。在 I-fetch 阶段的最后，指令和预测位都准备好了。预测位译码为 T（跳转）或者 U（不跳转）。如果指令不是分支指令，那么预测位直接丢弃。而如果指令是分支指令，那么预测位决定了下一条指令的地址。假如这个分支被预测为不跳转，那么会立即到 PC + 4 处取新指令。否则（分支被预测为跳转），在取下一条指令之前，首先需要得到分支的目标地址（这个问题会在后面解决。）

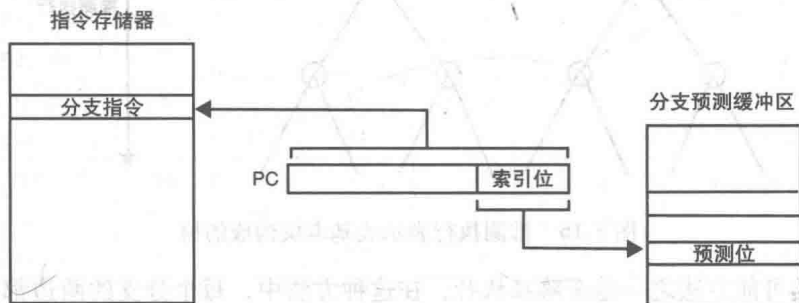


图 3-17 在 I-fetch 中同时访问指令和分支预测缓冲区

在分支执行完之后，分支预测结果将会得到确认，分支预测缓冲区可能会被更新。这个预测缓冲区一般比较小（1K ~ 250K 项），通常使用当前程序计数器的若干最低位来访问。多个分支之间可能出现重名，比如两个静态分支可能映射到同一个预测缓冲项，因此不同分支的预测结果可能会彼此冲突，这可能导致预测准确度很差。需要注意的是，这不是正确性问题，而是一个性能问题，并且可以通过增大缓冲区来改进。

预测器

最基本的动态分支预测器叫作一位预测器，每个预测缓冲区项有一个比特位，该位表明这个分支上一次执行的时候是否跳转。假如该分支上次跳转了，那么这次还会被预测为跳转，假如上次没有跳转，那么这次也预测为不跳转。换句话说，分支的预测结果和它上一次执行结果相同。

一位预测器主要针对的是控制一个循环体是否退出的条件分支。不管分支指令是在循环的头部还是底部，其结果在整个循环中不断重复且保持不变，直到最后一次迭代，而最后一次预测不可避免会失败。因此，每执行一个循环体，一位预测器会预测失败两次。

例 3.3 嵌套循环中的一位预测器失败率

考虑如下的带有嵌套循环的两个循环体：

```

Loop1: ---
      ---
Loop2: ---
      BEZ R2, Loop2
      ---
      BNEZ R3, Loop1
  
```

虚线表示任意指令。不管 loop 如何执行，loop2 的迭代次数都是 100。请问一位预测器的预测错误率是多少？

循环执行的时候，BEZ 在大多数情况下会跳转。不过，在最后一个迭代中，它不应该跳转，但是会被错误地预测为跳转。然后和这个分支相关的预测位会被重置。在第二次循环中，

该分支又会预测错误。所以 loop2 每次执行会导致 BEZ 预测错误两次。假如 loop2 执行 100 次迭代，那么错误预测的比率是 2%，这个错误率已经非常小了，但是在带有推测执行功能的处理器中，每次分支预测错误都会导致很多指令不得不被取消。

一位预测器的问题在于它对输出结果临时改变的反应过于迅速。假设一个分支在跳转和不跳转之间不断变化，那么一位预测器就总是会预测错误。即使这种模式其实是很容易预测的，而且用合适的预测器可以达到 100% 的正确性。

两位预测器提高了在分支控制循环上的预测正确率。其基本出发点是在两次连续的错误预测而不是一次错误预测之后，才改变预测方向。一个简单的两位预测器可以用 2 比特的饱和计数器来实现，每个计数器对应一个静态分支指令，如图 3-18 所示。分支一旦跳转，那么饱和计数器就加 1，如果分支未跳转，计数器就减 1，计数器在 0 和 3 的时候达到饱和。我们也可以为两位预测器设计出更加复杂的状态图，两位计数器的不同状态图可以正确预测不同的模式。两位预测器的主要贡献是把在循环中的错误分支预测数量减少到一位预测器的 50%。这是因为，当执行跳出循环的时候，预测仍然是跳转，但当再次进入循环的时候，分支会被正确预测。这个两位预测器也不能消除循环退出时的错误预测。我们还可以设计超过两位的计数器状态图。然而，在实际中，两位计数器几乎已经发掘了这一简单方法的所有潜能。

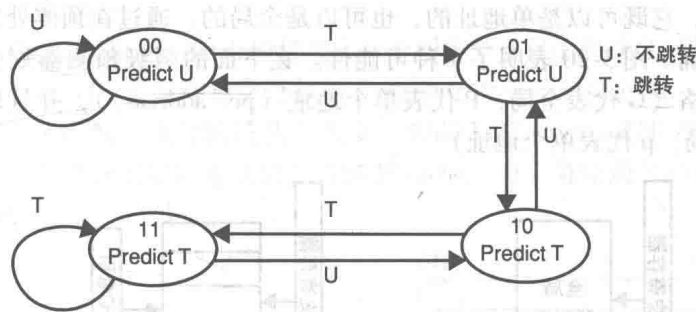


图 3-18 带饱和计数器的两位预测器状态转换图

关联分支预测器

另一个提高分支预测器的想法是以程序序跟踪先前分支的输出，不管它们是相同的还是不同的静态分支。原因是分支可能会依据到达它的执行路径的不同而有不同的表现行为。不仅如此，分支的输出也可能是高度相关联的，比如下面的例子：

```
if (a==2) then a:=0;
if (b==2) then b:=0;
if (a!=b) then ---
```

显然的，假如 a 和 b 都等于 2，那么第三个条件判断将会失败。不仅如此，假如 a = 2 且 b 不等于 2，那么第三个分支将有可能成功。最后，假如 a 和 b 都不等于 2，那么一切都不确定了。这种相关性无法被如图 3-17 那样的简单分支预测缓冲区所捕获，预测算法需要跟踪最近执行分支的结果。前 m 个条件分支的输出结果可以保存在 m 位的移位寄存器中，这也被称作全局分支历史寄存器。每当一个分支条件被确定时，其结果（未跳转（0）或者跳转（1））就会被移进这个寄存器。正如图 3-19 所示的那样，预测缓冲区通过 m 和 n 两个比特位的组合进行访问，其中 m 个比特位来自全局分支历史寄存器，另外 n 个位则来自程序计数器。另外，如果 m = n，那么分支预测缓冲区可以用 m 和 n 的按位异或结果来访问，这种情况下的预测器被称为 gshare。

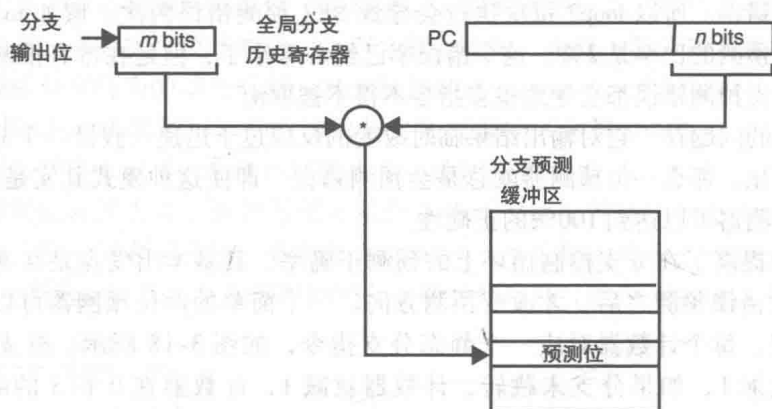


图 3-19 带有全局分支历史的分支预测缓冲区

两级预测器

关联分支预测器的成功，说明分支结果和先前的分支结果是相关的。这引出了两级分支预测器的通常形式：第一级是历史向量或者表格，它既可能是某条特定分支指令对应的之前的分支预测输出结果（单个地址的历史），也可能是所有分支的预测输出结果（全局历史）；第二级是预测表，同样，它既可以是单地址的，也可以是全局的。通过在预测处理中加入更多的信息，预测会更加准确。图 3-20 表明了 4 种可能性。在下面的两级预测器划分中，第一个字母表示历史向量或表格（G 代表全局，P 代表单个地址（per-address）），并且最后一个字母代表预测表（g 代表全局，p 代表单个地址）

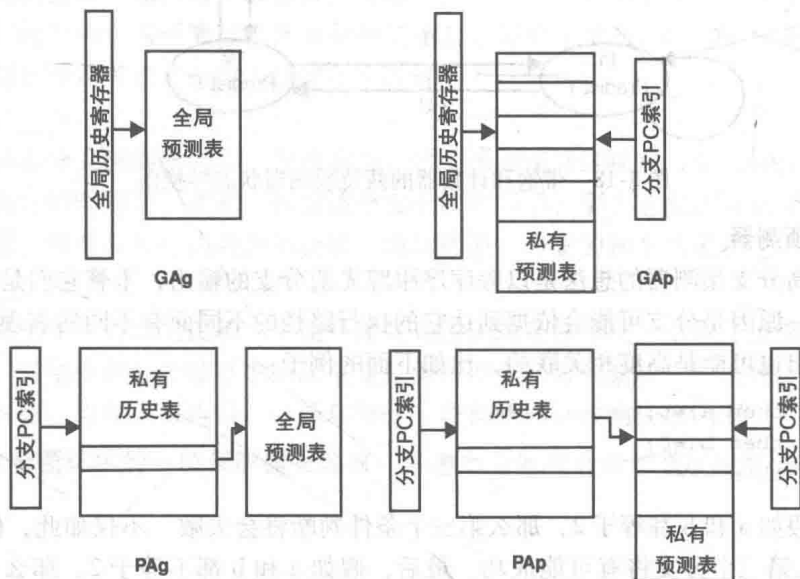


图 3-20 两级分支预测器

- Gag, 全局历史表和全局预测表。所有的分支都共享历史和预测器。具有相同全局历史的分支会相互干扰，并且通过相同的比特位预测。
- Gap, 全局历史表和单地址预测表。所有分支共享历史，但是对于每个分支来说，预测器是不同的。每个分支有自己的预测器比特位，这个比特位基于全局历史更新。这是图 3-19 所展示的预测器。

- PAg, 单地址历史表和全局预测器表。每个分支的历史单独记录, 但是预测器是共享的, 所以若所有的具有相同地址历史表的分支共享一个预测器, 会造成预测干扰。
- PAp, 单地址历史表和单地址预测器表。分支历史和预测器对于每个分支来说都是私有的。

组合预测器

分支预测器在每个分支上的表现并不完全相同。对于不同的程序或者一个程序的不同阶段, 不同预测器的表现也不同。因此为了进一步降低错误预测率, 多个预测器的预测结果经常通过一个选择器或者投票器组合起来。选择器追踪所有预测器的错误预测率, 并且基于它们的动态跟踪记录, 在任何时候选择其中的一个。在图 3-21 中展示了 3 个预测器的组合。

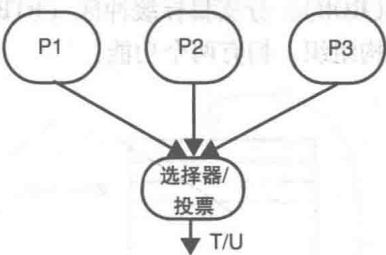


图 3-21 3 个预测器的组合预测

分支目标缓冲区

当分支被预测为不跳转时, 下一条指令的位置是 (PC) + 4。然而, 当分支被预测为跳转时, 在获取新指令之前必须确定目标地址。在分支指令取指之后的时钟周期内, 相应的目标地址会在加法器中计算出来。因此, 每次分支预测跳转之后, 取指队列的指令供给就会延迟一个时钟周期。

为了解决这个问题, 可以在 I-fetch 阶段加入一个叫作分支目标缓冲区 (Branch Target Buffer, BTB) 的小容量 cache, BTB 中保存着最近执行过的分支指令的目标地址。BTB 非常有效, 因为静态分支的目标地址在整个执行阶段从不改变。如图 3-22 所示, BTB 没有给分支地址引入别名, 它实际上被组织成分支目标地址的直接映射 cache。为了避免别名和执行不需要的指令, 所有的 PC 位都会被确认。

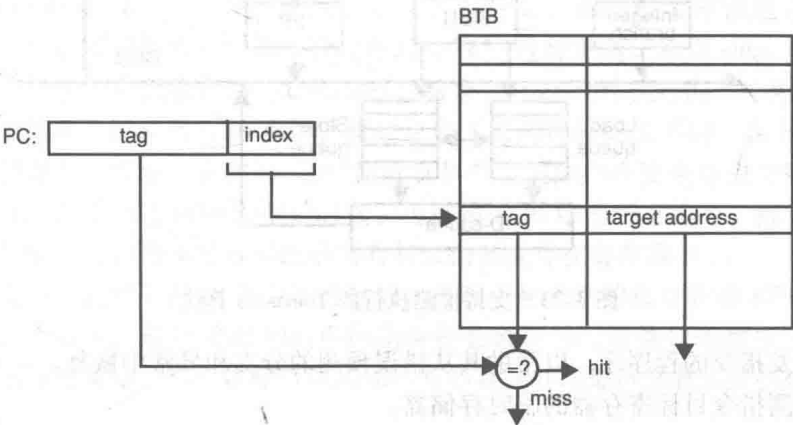


图 3-22 分支目标缓冲区 (BTB)

通过一个分支预测缓冲区 (Branch Prediction Buffer, BPB)、一个 BTB 以及在 I-fetch 阶段最后译码出的少量操作码, 就可以预测分支, 并且其目标地址在 I-fetch 阶段的最后就可以获知, 所以下一条指令在下一个周期就可以取指, 而不需要中断取指流。

3.4.4 支持推测的 Tomasulo 算法

为了支持跨条件分支的执行, 需要将下面的一些机制和资源加入到 Tomasulo 算法中: 分支预测、推测执行结果的临时存储, 以及从错误分支恢复的机制。另外, 还必须支持精确异常。

基本的出发点是推测地执行每一条指令，并推迟这条指令执行完成的影响，直到这条指令确定需要被执行才真正提交结果。因此指令还是被分发，执行，最终被提交，但是在指令提交之前，它一直是推测性的，并且其执行过程可以被回滚。如果当前指令之前的分支被预测错误或者之前的指令引发了一个异常，那么推测执行的指令就可能回滚。

在之前的非推测执行架构中加入这些单元后，新架构如图 3-23 所示：重排序缓冲区 (ROB)，分支目标缓冲区 (BTB)，分支预测缓冲区 (BPB)。ROB 以先进先出 (FIFO) 的结构组织，拥有两个功能：

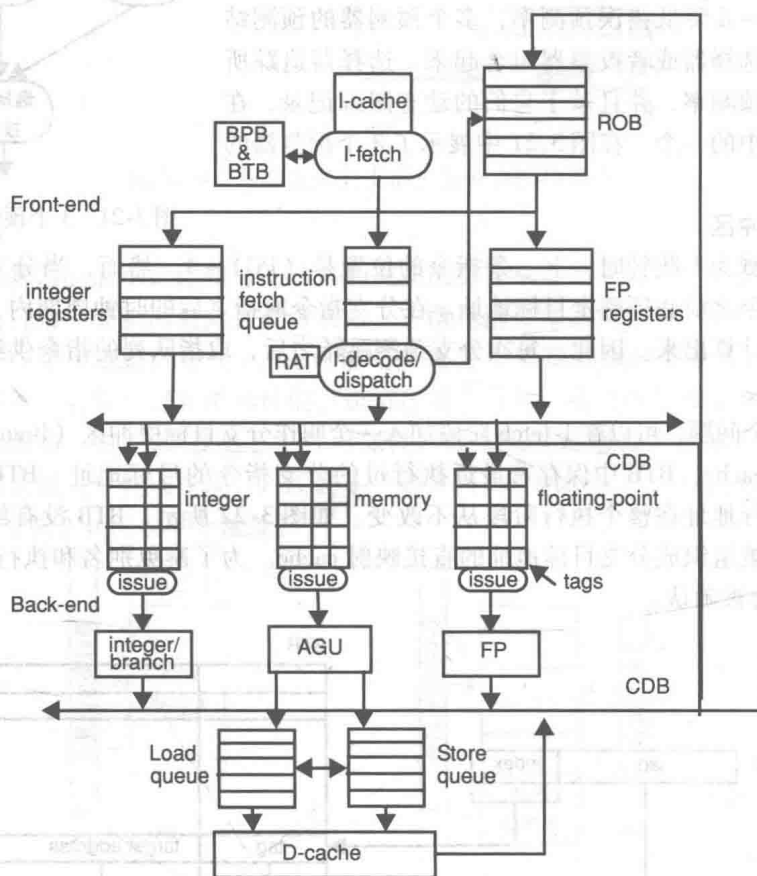


图 3-23 支持推测执行的 Tomasulo 算法

- 记录分发指令的程序序，以帮助其从错误预测的分支和异常中恢复。
- 作为推测指令目标寄存器的临时存储器。

指令按照程序序取指并且以程序序在取指队列中移动。当 I-fetch 阶段取到一个分支指令时，它在 BPB 中进行预测，而目标地址则从 BTB 中获取。

分发单元为每条新指令分配一个 ROB 项和一个发射队列项。假如是 load 或者 store 指令，还会另外分配一个 load/store 队列项。如果上述结构中某一个已经满了，那么分发单元必须暂停。分发单元的第二个作用就是管理寄存器重命名。分发单元跟踪寄存器值在寄存器别名表 (register alias table, RAT) 中的位置。一个整型或者浮点型寄存器的值可能后端等待，也可能在 ROB 中推测执行，还可能在寄存器堆中提交。结构寄存器通过寄存器号访问 RAT，RAT 中的每个项指向对应寄存器最新的值所在的位置 (最近修改的指令对应的标识 (tag) 或者是寄存器号)，外加一个位标志，表示其值是否在后端等待 (未就绪)。寄存器通过 ROB 中生成

该寄存器值的指令位置进行重命名。对于每条进入的指令，分发单元会将目标寄存器重命名为分配给这条指令（tag）的 ROB 项，并且检查 RAT 中输入操作数的状态，如果操作数可用（在 ROB 中或者寄存器堆中），分发单元就将这个值传到发射队列（操作数就绪）。若操作数在后端等待中，分发单元就发送分配给这条指令的 ROB 项编号（操作数未就绪）。

从分发开始到提交，ROB 中的项会保存每条推测执行指令的记录。每个 ROB 项包含若干个数据域：指令类型、目的寄存器号、目的寄存器值和完成标志位。目标寄存器值会被指令的结果值填充，而指令完成之后，会设置其完成标志位。

ROB 被组织成一个环形的 FIFO 缓冲区，所以通过 ROB 项编号，每个结果可以直接写入指令项中，也可以很容易刷掉一部分缓冲区。在图 3-24 中，（新的）分发指令被分配到“bottom”指向的 ROB 项中，被“top”指向的指令项则可以退出（retire）/提交。bottom 指针只需要简单地向上移动一下就可以刷掉最新的指令。

和 Tomasulo 算法一样，当操作数全部可用的时候，发射队列中的指令就开始执行。当执行完成之后，指令把结果写到公用数据总线上，在发射队列中等待这个结果的项捕获这个结果值，结果值被写入 ROB 项，分发单元再将 RAT 中相应的值修改成“可用并且是推测的”（available and speculative）。需注意的是，这里的值通过 ROB 的项号直接写入 ROB。寄存器不用像 Tomasulo 算法那样保持标识（tag）。当指令到达 ROB 的顶部时，寄存器就会更新。

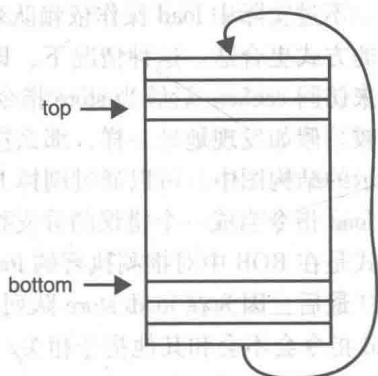


图 3-24 环形 FIFO 缓冲实现的 ROB 结构

在 ROB 顶部，推测执行的指令按照程序序退出（提交）。这个时候推测指令已经成为确定的指令了（non-speculative），因为所有的指令（包括分支）都已经无异常地完成。在退出之前，存储单元（比如寄存器或者需要访存时候的主存）进行更新。所以 store 指令只有在到达 ROB 顶部的时候才会访问数据存储（D-cache），这是对 store 操作执行的严格限制。

分支的输出结果会记录到 ROB 项中，假如它和之前的预测结果不同，在 ROB 中所有该分支之后的指令都要进行回滚（刷掉），同时该分支指令之前的指令要全部提交完成。比较方便的做法是等到分支指令到达 ROB 顶部的时候，再刷掉 ROB 中的所有指令，流水线后端的内容也要全部刷掉。RAT 可以简单设置成当前所有有效值都保存在寄存器中。

异常在触发的时候首先在 ROB 中进行标记，直至引起异常的指令到达 ROB 顶部的时候才会真正处理。这个时候，整个 ROB 和后端的指令都会被刷掉，异常处理程序才会开始。这和五级流水线中用过的策略一样，这样可以确保所有的异常处理都依据进程序执行。

3.4.5 动态内存歧义消除

Tomasulo 算法中的技术适合用来解决访存相关。就像之前介绍的，store 指令在分发的时候可以分割成两个子指令：一个在 AGU 中计算地址，一个传送数据到 store 队列。

load/store 队列保持 load 和 store 指令的程序序，解决访存的所有相关性。当 store 的地址和数据都已经在 load/store 队列中就绪时，我们认为 store 已经完成（执行完成）。已经完成的 store 指令在 load/store 队列中等待提交，当它到达 ROB 顶部的时候就可以提交结果到 cache，并且在下一个周期退出。访存操作数上的 WAW 和 WAR 相关可以自动解决，因为 store 指令只有在到达 ROB 顶部的时候，指令才能更新 cache，所以在 store 更新 cache 的时候，它之前的 load 指令和 store 指令都已经提交了。

不过，RAW 相关则必须检查。主要有两种方法：保守方法和乐观方法。在保守方法中，当 load/store 队列中的 load 指令 and 所有之前在 load/store 队列中等待的 store 指令的地址都已经明确之后，就可以访问 cache。若队列中先于某条 load 指令的 store 指令与之有相同的访存地址，那么 load 指令必须等待该 store 指令在 cache 中执行之后才能发射执行。在某些特殊的情况下，load 可以执行返回 load/store 队列中最新 store 的数据，前提是它们的地址相同，并且 store 的数据是已知的。假如之前的 store 地址是未知的，那么硬件必须考虑到最坏的情况（store 和 load 的地址相同），这时，load 必须等待 store 的地址明确之后再检测它们是否相同。

不过实际中 load 操作依赖队列中之前 store 指令结果的情况是很少见的，因此可能采用乐观的方式更合适。这种情况下，即使之前的 store 指令地址还不知道，load 指令也可以推测执行来访问 cache。后续当 store 指令的地址确定时，会再和刚刚完成的那个 load 指令的地址进行比较。假如发现地址一样，那么这个 load 指令和其后的所有指令都需要重新执行。在图 3-23 所示的结构图中，可以通过刷掉 ROB 中 load 及其后续指令的方式进行重放，这种情况就好像把 load 指令当成一个错误的分支指令一样，再从 load 指令开始重新取指。实现这一机制的便捷方式是在 ROB 中对推测执行的 load 打上标记并等到 load 指令到达 ROB 顶端进行处理。

最后，因为在 load/store 队列中的 RAW 相关导致的回滚代价很高，所以有必要提前预测 load 指令会不会和其他指令相关。假如这条 load 指令相关的可能性很大，那么就采用保守的方式，否则采取乐观的方式。

例 3.4 支持推测的 Tomasulo 算法下的执行过程 我们按照逐个周期执行的方式跟踪例 3.2 中的一段代码。在完成表 3-15 时，例 3.2 中的假定同样适用，不同之处如下：

- 增加一个新的退出（retire）流水级，指令完成之后顺序退出。
- stores 指令必须到达 ROB 的顶部才能更新 cache。store 指令在 load/store 队列中等待。注意，一旦 store 指令的地址和数据就绪了就可以执行，计算 store 地址的子指令不会单独分配 ROB 项。
- 支持分支预测，因此分支的后续指令可以推测分发。

表 3-15 支持推测的 Tomasulo 算法的执行示例

		分发	发射	开始执行	完成执行	cache	CDB	retire	备注
I1	L S F0, 0(R1)	1	2	(3)	3	(4)	(5)	6	
I2	L S F1, 0(R2)	2	3	(4)	4	(5)	(6)	7	
I3	ADD. S F2, F1, F0	3	7	(8)	12	—	(13)	14	等待 F1
I4	S. S-A F2, 0(R1)	4	5	(6)	6	—	—	—	
I5	S. S-D F2, 0(R1)	5	14	(15)	(15)	(16)	—	17	等待 F2
I6	ADDI R1, R1, #4	6	7	(8)	8	—	(9)	18	
I7	ADDI R2, R2, #4	7	8	(9)	9	—	(10)	19	
I8	SUBI R3, R3, #1	8	9	(10)	10	—	(11)	20	
I9	BNEZ R3, Loop	9	14	(15)	15	—	(16)	21	等待 R3 的值， CDB 流水级中和 I10 以及 I11 存 在冲突
I10	L S F0, 0(R1)	10	11	(12)	12	(13)	(14)	22	
I11	L S F1, 0(R2)	11	12	(13)	13	(14)	(15)	23	
I12	ADD. S F2, F1, F0	12	16	(17)	21	—	(22)	24	等待 F1

在表 3-15 中, I5 store 指令通过 AGU 阶段, 在第 15 个时钟周期的末尾, store 的数据被放入 load/store 队列。这时, store 指令的地址也在 load/store 队列中, 事实上, 从第 7 个时钟周期开始, 地址已经得到了。这对之前的访存操作没有影响, 因为此时前面的访存操作都已经完成退出了。store 指令可以在下一个时钟周期 (时钟 16) 更新 cache, 因为之前 (进程序) 的 I3 指令在时钟 14 的时候已经退出, 所以从时钟 14 开始 store 指令到达 ROB 的顶部。然后 Store 指令跳过 CDB 总线阶段, 在下一个阶段 (时钟 17) 退出, 最后 I10 load 指令的地址在第 12 个时钟周期的末尾写入 load/store 队列。之前 I5 对应的 store 指令一直在等待, 但是从第 7 个时钟周期开始, 它的地址就已经是确定的了, 并且和 I10 load 指令的地址不同, 因此 load 指令可以在第 13 个时钟周期访问 cache, 并且绕过之前的 store 指令。

在表 3-15 中, I9 和 I10 的调度过程显示, 假如不采用逐个时钟周期, 而是采用逐行的方式填充这个调度表格的话, 是很危险的。假如表格用逐行的方式填充, I9 指令会在 R3 寄存器的值在 CDB 总线上传播 (在第 12 个时钟周期) 之后立刻发射, 并且 I10 和 I11 指令会被延迟。然而事实上, I10 指令在 I9 指令之前就已经就绪了 (在第 11 个时钟周期), 所以 I10 一定会像表中的那样先发射。这种调度的不同也会影响指令 I11 和 I12 的调度。

本例中的执行时间比例 3.4 的执行时间要短, 这主要是因为分支之后的 load 指令不需要等到分支完成后才进行分发。

3.4.6 显式寄存器重命名

目前, 我们介绍的 ROB 既可以控制指令的提交顺序, 又可以用来支持寄存器的重命名。然而, 也可以让 ROB 只简单地用于追踪指令的执行顺序, 而把寄存器重命名的功能放在物理寄存器堆中显式完成, 而不再依赖于 ROB。

物理寄存器的数量比结构寄存器 (也就是指令集可访问到的寄存器) 的数量多。结构寄存器动态映射到物理寄存器, 在任意时刻, 一个结构寄存器可能需要多个物理寄存器来保存多个推测的值。一些物理寄存器需要保持结构寄存器已经提交的数据, 这样做是为了预防异常的出现。另一些物理寄存器则需要用来保持结构寄存器最新的值 (推测执行或者非推测执行的最新值)。

从结构寄存器号到物理寄存器号的映射是通过两个寄存器别名表 (RAT) 的映射机制完成的。第一个映射表 (前端 RAT) 记录了每个结构寄存器到包含其最新值 (有 1 位表示这个值是否还在流水线后端等待结果) 的物理寄存器的映射关系, 而第二个映射表 (retirement RAT) 则记录了结构寄存器到包含它最新提交值的物理寄存器的映射关系。这两个映射表的指针可能会一样, 在这种情况下, 结构寄存器最新的值也是其最近提交的值。

当一条写寄存器指令到达分发阶段时, 将会从空闲寄存器链表中分配一个新的物理寄存器来保存目标结构寄存器的新值, 前端 RAT 会查表来获取输入寄存器操作数的最新值 (假如这个值没有在流水线后端等待)。如果值没有在后端等待, 物理寄存器组中的值就会被发送到发射队列; 否则, 就会把物理寄存器号 (对应之前介绍机器中的 tag) 发送到发射队列, 并且将操作数域置成未就绪。如果空闲链表为空, 分发单元就会暂停。当某个 retired 值被对应的同一个结构寄存器的另一个 retired 值覆盖之后, 带有旧的 retired 值的物理寄存器就会被释放, 并加入到空闲链表。从计算这个值的指令被分发开始, 一直到对应相同结构寄存器的另一个新的物理寄存器 retire 时, 整个过程中, 物理寄存器都是分配给这个值的。

图 3-25a 给出了一个带有 32 个结构寄存器和 128 个物理寄存器的例子。在这个例子中, 3 个物理寄存器被分配给结构寄存器 r20, 一个是存储最近的 (推测的) 值, 另一个存储最近的 retired 值, 最后一个存储另一个推测值。一旦计算推测值的指令完成 retired, 这个推测值就变

3.4.7 指令发射后的寄存器读取

在分发阶段读取寄存器的值使得指令的分发和退出变得更加复杂。退出变得复杂是因为寄存器的值必须物理上从 ROB 写入结构寄存器组中（图 3-23）。在支持显式寄存器重命名的机器上，在指令退出时，只有很少量的 retirement RAT 需要进行更新。

在最新的微架构中，分发阶段通常不读取输入寄存器操作数。所有的输入寄存器值在指令从发射队列发射到功能单元之后才被读取。在这种方法中，前端大部分是没有变化的，只有分发单元里的重命名逻辑需要把输入寄存器的物理寄存器号发送给发射队列（并且加上一个就绪位）。当指令在 CDB 总线上完成的时候，它将在发射队列中等待指令的就绪位进行置位。此后选择就绪指令发射到功能部件。被发射指令首先访问寄存器堆来获得全部的输入操作数，然后输送到对应的功能部件。图 3-26 给出了这种新的微架构。

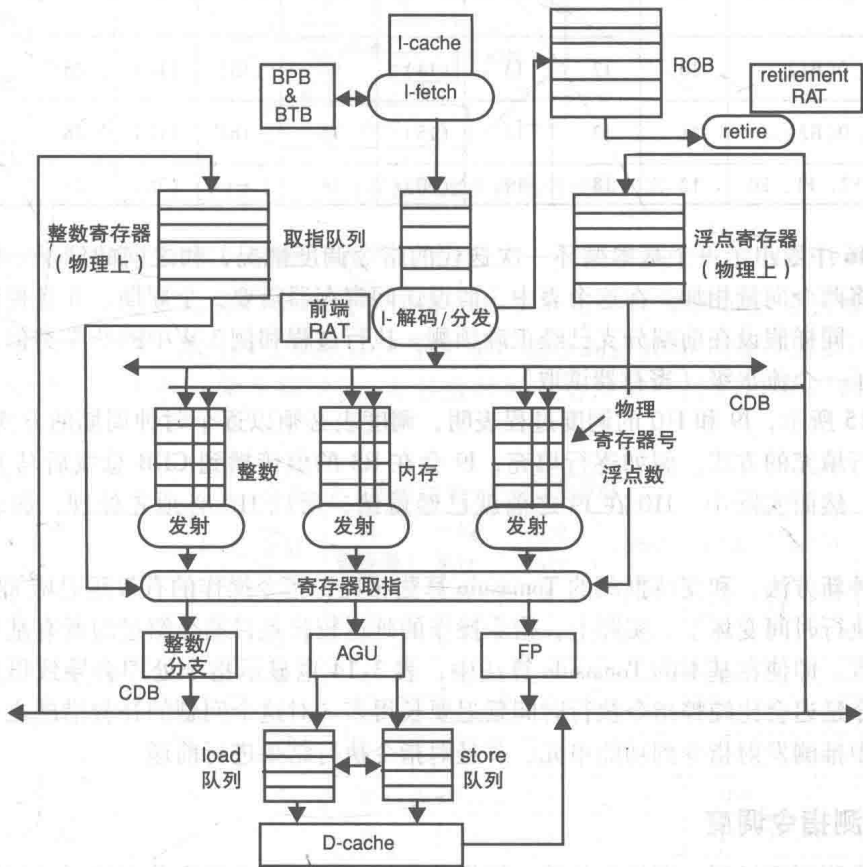


图 3-26 支持显式寄存器重命名和发射后寄存器读取的推测微架构

例 3.5 支持发射后寄存器读取的执行过程 假设使用例 3.4 中的相同的参数和代码，我们来完成表 3-16。假设寄存器堆有足够的带宽支持无冲突地读写寄存器。

表 3-16 执行示例——支持预测和发射后寄存器读取

		分发	发射	读寄存器	开始执行	完成执行	cache	CDB	retire	备注
I1	L S F0, 0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L S F1, 0(R2)	2	3	4	(5)	5	(6)	(7)	8	

(续)

		分发	发射	读 寄存器	开始 执行	完成 执行	cache	CDB	retire	备注
I3	ADD. S F2, F1, F0	3	8	9	(10)	14	—	(15)	16	等待 F1
I4	S. S-A F2, 0(R1)	4	5	6	(7)	7	—	—		
I5	S. S-D F2, 0(R1)	5	16	17	(18)	(18)	(19)	—	20	等待 F2
I6	ADDI R1, R1, #4	6	7	8	(9)	9	—	(10)	21	
I7	ADDI R2, R2, #4	7	8	9	(10)	10	—	(11)	22	
I8	SUBI R3, R3, #1	8	9	10	(11)	11	—	(12)	23	
I9	BNEZ R3, Loop	9	15	16	(17)	17		(18)	24	等待 R3 的 值, CDB 流水 级中和 I10 以 及 I11 存在 冲突
I10	L S F0, 0(R1)	10	12	13	(14)	14	(15)	(16)	25	CDB 与 I3 冲突
I11	L S F1, 0(R2)	11	13	14	(15)	15	(16)	(17)	26	FU 与 I10 冲突
I12	ADD. S F2, F1, F0	12	18	19	(20)	24	—	(25)	27	等待 F1

在表 3-16 中给出了一个基本循环一次迭代的指令调度情况, 和之前的例子一样, 指令序列的功能是将两个向量相加。在这个表中, 假设访问寄存器需要一个周期, 并且控制指令没有额外的开销。同样假设在前端分支已经正确预测。执行过程和例 3.4 中的非常类似, 只是在发射之后增加了一个流水级 (寄存器读取)。

如表 3-15 所示, I9 和 I10 的调度过程表明, 调度表必须以逐个时钟周期的方式进行填充, 而不能按逐行填充的方式。假如逐行填充, I9 会在 R3 的值传播到 CDB 总线后马上发射, I10 则会被延迟。然而实际中, I10 在 I9 之前就已经就绪, 所以 I10 必须先处理, 如表中所示的那样。

通过这种新方法, 和支持推测的 Tomasulo 算法相比, 指令操作的有效延迟增加了 1 个时钟周期, 所以执行时间变坏了。实际上, 指令操作的延迟较长是目前介绍过的所有乱序处理器结构的共同弱点。即使在基本的 Tomasulo 算法中, 表 3-14 也显示指令处理会导致很多额外的开销, 所以指令延迟会比纯粹指令执行时间延迟要长得多。对这个问题的补救措施之一是, 可以从发射队列中推测发射指令到功能单元, 并且对指令执行结果进行前递。

3.4.8 推测指令调度

借助于推测调度机制, 在指令的输入操作数可用之前, 指令调度单元就可以预测指令什么时候发射。由于大多数指令的操作延迟是确定的, 因此, 指令调度单元可以在父指令发射并且操作延迟满足之后, 马上调度它的子指令。在推测调度的决定中, 像公用数据总线这样的资源冲突也需要考虑进来。如图 3-27 所示的那样, 后端需要稍加修改。不再由数据总线来唤醒操作数, 而是由和发射逻辑相关的推测指令调度单元来触发唤醒操作。不但如此, 寄存器的值也可以直接前递给进入功能单元的子指令, 因为子指令已经被推测调度, 此时刚好可以获取到父指令的结果。这个前递逻辑和带有浮点单元的静态五级流水线非常相似 (见图 3-8)。这种调度机制可能会在指令输入操作数就绪和指令运行结果已经发送到数据总线上这两个时刻之间引入更多的流水级。不过, 只要推测调度能够成功, 那么增加的延迟就不会影响性能。

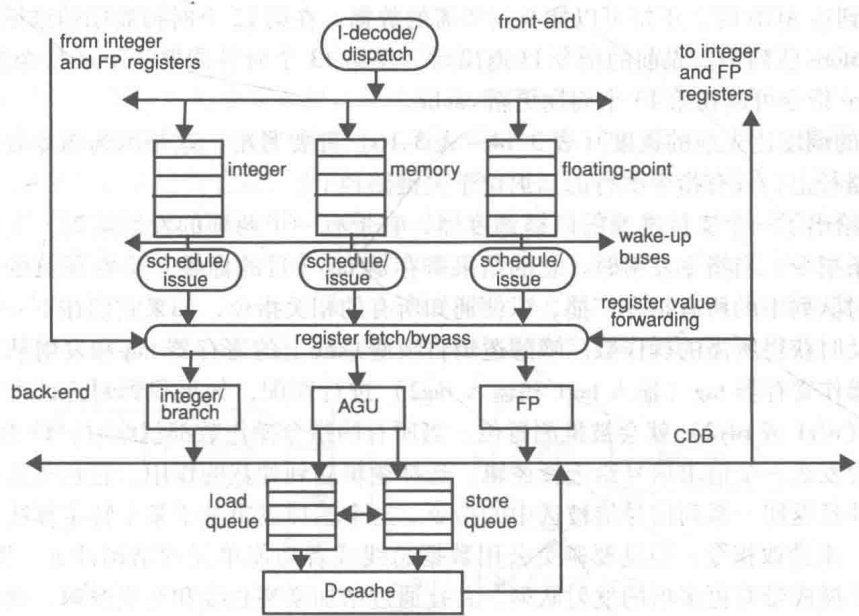


图 3-27 支持推测调度的推测微架构后端

例 3.6 推测调度下的执行过程 表 3-17 给出了一个推测调度机器的执行调度结果，和例 3.5 使用相同的代码和假设。在本例中，指令基于父指令的操作延迟进行推测发射，在 load/store 队列中处理内存地址歧义消除以及等待 store 指令到达 ROB 顶端。这里的调度假设所有的调度推测都是正确的（常见情况）。大多数时候，结果都通过 CDB 及时前递给相关的指令。

表 3-17 执行示例——推测调度

		分发	发射	读寄存器	开始执行	完成执行	cache	CDB	retire	备注
I1	L S F0, 0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L SF1, 0(R2)	2	3	4	(5)	5	(6)	(7)	8	
I3	ADD. S F2, F1, F0	3	5	6	(7)	11	—	(12)	13	等待 F1
I4	S. S-A F2, 0(R1)	4	5	6	(7)	7	—	—	—	
I5	S. S-D F2, 0(R1)	5	10	11	(12)	12	(13)	—	14	等待 F2
I6	ADDI R1, R1, #4	6	7	8	(9)	9	—	(10)	15	
I7	ADDI R2, R2, #4	7	8	9	(10)	10	—	(11)	16	
I8	SUBI R3, R3, #1	8	10	11	(12)	12	—	(13)	17	CDB 与 I3 冲突
I9	BNEZ R3, Loop	9	11	12	(13)	13	—	(14)	18	issue 与 I8 冲突
I10	L S F0, 0(R1)	10	11	12	(13)	13	(14)	(15)	19	
I11	L S F1, 0(R2)	11	12	13	(14)	14	(15)	(16)	20	
I12	ADD. S F2, F1, F0	12	14	15	(16)	20	—	(21)	22	等待 F1

在表 3-17 中，指令 I3 在第 5 个时钟周期发射，以便在指令开始执行时（第 7 个时钟周期）接收指令 I1 的前递结果。I3 在 I1 load 指令之后 2 个时钟周期推测执行，因为 load 指令的延迟是 2 个时钟周期（AGU + cache 访问）。store I5 在 ADD. S 指令之后的 5 个时钟周期推测发射，

这样当 store 到达 AGU 时，正好可以捕获到所需的数据。在第 12 个时钟周期的结尾，store 操作锁存到 load/store 队列中。先前的指令 I3 退出后，在第 13 个时钟周期，store 指令到达 ROB 顶端，因此 store 指令可以在第 13 个周期更新 cache。

表 3-17 的调度比先前的调度（表 3-14 ~ 表 3-16）都要紧凑，这是因为指令处理的开销不再位于关键路径上，只有指令执行的延迟位于关键路径上。

图 3-28 给出了一个支持推测的广播调度器，它带有一个两项的发射队列，支持每个时钟周期发射一条指令。当指令发射时，它的结果寄存器 tag（目的标签）会在预测的指令操作延迟之后向发射队列中的所有指令广播，以便通知所有的相关指令，如果它们在下一个周期进行调度，可以及时获得所需的操作数。唤醒逻辑将唤醒总线上的寄存器 tag 和发射队列中所有项的两个输入操作寄存器 tag（输入 tag1 和输入 tag2）进行匹配，如果找到对应匹配，匹配操作数的就绪位（rdy1 或 rdy2）就会被推测置位。当所有的指令操作数都就绪时，该指令也就就绪了，然后它会发送一个请求信号给选择逻辑。选择逻辑起到仲裁的作用，它负责选择下一条执行的指令，并且返回一系列信号给被选中的指令。这个选择逻辑基于某个特定算法（比如最老指令优先法）来选取指令，但是要避免公用数据总线或者功能单元的结构冲突。图 3-28 中的调度器可以扩展成带有更多项的发射队列，并且通过增加唤醒总线 and 对应逻辑，调度器也可以扩展到在一个时钟周期内发射多条指令。

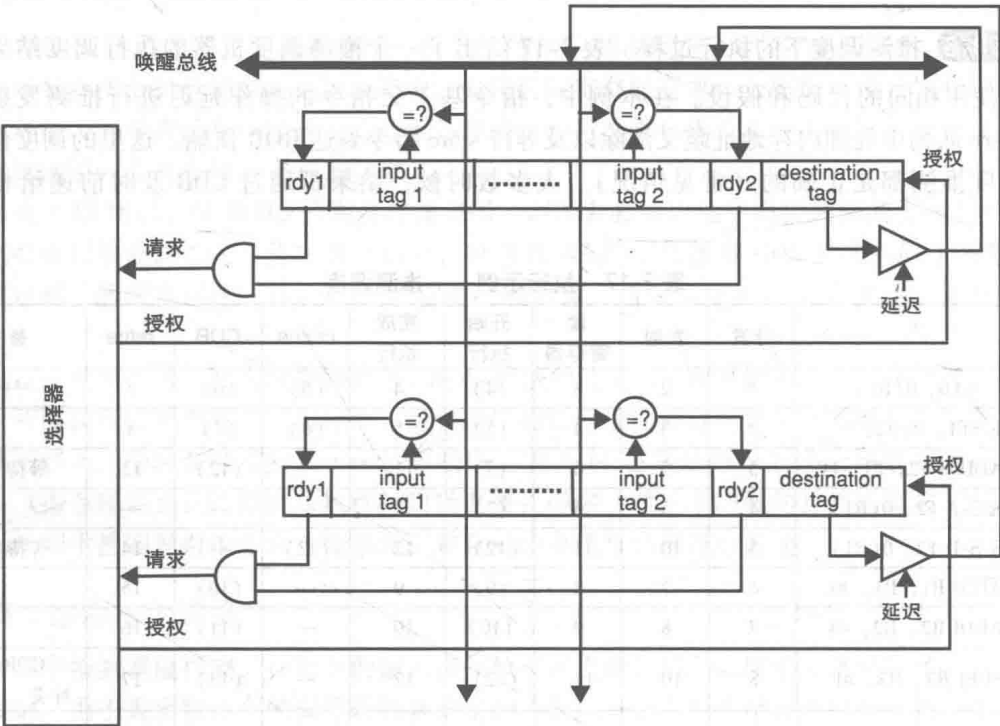


图 3-28 带有两个队列项的广播指令调度器

利用推测调度，多条已发射的相关指令可以同时送往功能部件。推测调度单元确保在父子指令之间的流水线时钟间隔至少是父操作的延迟。如果所有指令的操作延迟都在调度时可预测，那么这个方法就非常有效。而且，除了 load 指令之外，在其他大多数情况下确实都是这样的。大多数情况下 load 指令会命中 L1，所以延迟就是 L1 的访问时间。然而，假如 load 指令没有命中 L1，值返回的时间就被延后了，从而依赖 load 结果的指令就会错误调度。此时调度单元必须进行修正。这个修正过程可以通过 I-cache 和 ROB 来重放 load 和其后的所有指令（这里

的 load 指令就如同预测错误的分支指令一样处理)。

通常,相比于从 ROB 中重放,直接从发射队列中重放指令可以设计出更轻量级的重放机制。图 3-29 显示了一个简单的重放机制,每个物理寄存器有一个满/空 (F/E) 标志位,目标寄存器的 F/E 标志位在指令分发的时候进行重置,当指令在功能部件 (FU) 执行完,写入或前递寄存器值时会对 F/E 进行置位;当指令离开调度器时,它会被拷贝到重放队列中。当指令到达寄存器阶段 (同时由校验器 (图中的 checker) 进行校验) 时,如果指令输入寄存器的 F/E 标志位是重置状态,说明这条指令被错误调度了,它会被循环回去重复执行。指令经过校验后会被丢弃。错误调度的指令不会更改它们目的寄存器的 F/E 位,所有错误调度指令都按照图 3-29 中的指令链进行传播,这个指令链取决于第一条被错误调度的指令。

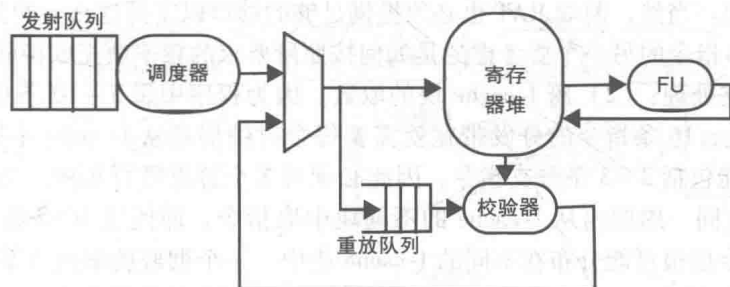


图 3-29 带有重放队列的预测调度器

3.4.9 打破数据流限制：值预测

即使支持推测调度,相关的指令也必须等待它的父指令延迟满足之后才能被调度。所以,想在输入操作数准备好之前就执行指令,看起来似乎是不可能的。计算过程可以映射成一张数据流图,在这张图中,每个节点是一条指令,指令延迟用一条连接父子指令的弧线表示。每一条弧线标记上父指令的执行延迟。由于指令必须等待其输入准备好,因此一次计算的最小执行时间由数据流图中关键路径上总的执行延迟所决定,这就是所谓的数据流限制。

研究表明许多指令的结果是高度可预测的。许多静态指令常常返回相同的常量值或者一个较小数据集上的某个值,其他一些指令 (例如,更新地址寄存器的指令) 也常常只是给输入值增加一个固定的常量。如果一条指令的输出值是可预测的,那么可以通过对子指令进行输入值预测并且立即调度来推测性地消除父指令和子指令之间的依赖关系。当然,父指令也必须执行完毕,这样才能验证预测结果的正确性。如果预测失败,子指令和它的所有相关指令必须重放执行,重放可以通过 I-cache 和 ROB 的支持来实现,或者使用像图 3-29 中那样的轻量级重放机制。对某些可能导致 cache 失效的 Load 指令进行值预测,对于缩短数据流图中关键路径的长度是非常重要的。由于值预测错误会造成较高的回退开销,因此值预测应当仅在具有高时延和值高度可预测性的指令中使用,静态指令的值可预测性可以通过它的动态命中率来记录。

值预测很容易,并且可以很高效地在图 3-23、图 3-26、图 3-27 所示的任意一种支持推测的微结构上下文中实现。接下来的例子将解释值预测是如何在支持推测调度机制的处理器中实现的。

例 3.7 支持推测调度的乱序执行处理器中的值预测 如图 3-27 中所示的支持推测调度微结构,利用指令的程序计数器,可以在流水线取指阶段查找值预测表,然后返回该指令执行结果的预测值。在指令分发阶段,预测值被存储在物理寄存器中,并且被标记为就绪状态。后续相关指令可能在被预测指令之前调度执行。当被预测指令执行完毕、结果写回寄存器时,首先

要和寄存器中的预测值做比较。如果值相等,继续执行;否则,这条被错误预测的指令的后续所有相关指令必须重新执行,简单的重新执行机制可以通过 I-cache 和 ROB 进行支持。◀

3.4.10 单周期多指令

图 3-23 和图 3-27 所示的结构中,每个时钟周期调度多条指令的主要困难在于需要在处理器的前端重命名多条相关指令。“重命名一条指令”意味着重命名指令的输入寄存器操作数和输出寄存器操作数。指令必须按照线程序调度。重命名相关指令的过程是固有串行的。在最坏情况下,重命名阶段必须为在同一时钟周期中分发的每条指令访问 3 个前端 RAT 项。要做到这一点,可以提高重命名流水级的频率,或者采用复杂的组合逻辑来处理每个时钟周期内的相关性和重命名实现。当然,前端 RAT 也必须提供足够的端口以支持这么大的分发带宽。

实现单周期多指令的另一个要考虑的是如何按照所要求的频率填充取指队列。主要有两个问题:(1)多分支处理;(2)跨 I-cache 块的取数。因为程序中每 5~10 条指令中平均就有 1 条分支指令,因此,16 条指令的分发带宽就需要每个时钟周期从 I-cache 中取 16 条指令,这 16 条指令中很可能包括 2~3 条分支指令,因此必须对多个分支进行预测。当这些分支预测为跳转时,就需要在同一周期内从 I-cache 的不同块中取指令。即使这 16 条指令中没有分支指令,16 条连续指令也很可能分布在不同的 I-cache 块中。一个彻底的解决方案是使用指令踪迹 cache 来替换传统的指令 cache。不同于静态存储顺序保存指令的普通指令 cache,trace cache 保持指令(可能已经被译码)的动态轨迹。与普通 I-cache 不同,一次取指可以从 trace cache 中按照动态线程顺序读取连续的一串指令。

最后,流水线后端也必须仔细设计以确保没有其他瓶颈。比如,必须提供足够的功能单元用于执行操作以便尽可能降低动态指令执行时的结构冲突。此外,可能还需要支持多个 CDB,如果只有一个 CDB 的话,指令吞吐量就会被限制为每个时钟周期只能有一条写寄存器结果的指令。最后,寄存器文件和 retirement 流水级也必须有足够的带宽,以便能够处理每个时钟周期内的多条指令。

3.4.11 处理复杂 ISA

目前为止,我们都假定的是一个简单的 RISC 指令集。不过,复杂指令集也同样可以利用乱序执行和推测执行的优势,x86 架构的奔腾Ⅲ和奔腾 4 (NetBurst) 微结构已经证明了这一点。和之前一样,需要多个时钟周期的复杂指令也必须以微指令方式执行而不是直接在硬件中执行。并且,在支持微指令方式时不能影响最频繁执行的那部分指令集核心指令(类 RISC 指令)。

一个可行的实现技巧是使用那部分核心的类 RISC 指令直接作为微指令来执行其他复杂指令。在奔腾Ⅲ和奔腾 4 微结构中,x86 指令被划分为三类:第一类是类 RISC 指令,只有一条微操作(uop),可以在一个时钟周期里执行;第二类是简单指令,可以被翻译为最多 4 条微操作;第三类是复杂指令,需要翻译为 4 条以上的微操作。译码之后微操作立即被送到微操作队列。需要翻译为最多 4 条微操作的指令可以通过译码器即时翻译。那些需要被翻译为多于 4 条微操作的指令则需要跳转到一段微代码序列中,这段微代码序列存储在指令译码器可访问的微存储中。这种结构下的前端需要做轻微修改,如图 3-30 所示。

作为 ISA 核心部分的类 RISC 指令可以一对一地映射到微操作上,执行速度很快,不会影响译码的时间。类 RISC 指令可以在一拍内完成译码,同样,需要翻译成 2~4 条微操作的简单指令也可以在一拍内完成译码。而其他复杂指令的译码开销更大,因为它们需要从微指令存储中执行,不过这些开销可以通过较长的微操作序列进行分摊。由于这些复杂指令非常少,它们对性能几乎没有影响。

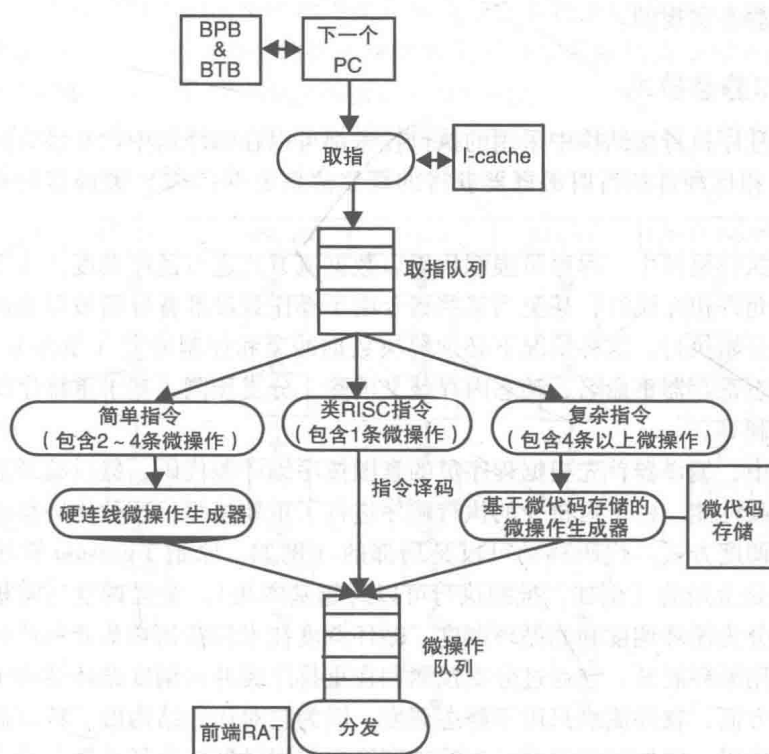


图 3-30 复杂 ISA 的前端

复杂指令集中，I-fetch 阶段不再总是一次取整条指令，因为指令是变长的。一条指令可能跨越两条以上的 I-cache 行边界，从而在取指令流水线中产生空泡。分支或者跳转指令也可能重定向执行到代码中的任意字节处。此外，下一条指令的地址（下一个 PC）计算也不是那么容易确定。为了解决上述问题，我们在 I-fetch 阶段（如图 3-30 所示，I-fetch 阶段从 I-cache 中取指）之前增加一个计算下一个 PC 的流水级。这个新的流水级基于当前 PC 值、当前指令长度和 BPB、BTB 的内容预测下一条指令地址。PC 寄存器现在指向内存中的一个字节或者 I-cache 行中的一个字节，PC 地址对应的是内存地址或者下一条指令首字节在 cache 中的物理位置。译码单元处理来自 IFQ 队列的指令字节，然后尽可能快地生成对应的微操作。译码逻辑必须对来自 IFQ 的指令进行对齐，并且在一个时钟周期内完成译码。之后，译码后的微操作被移入微操作队列。

相比于 RISC 流水线，上述方案中的流水线在某种程度上被拉长了，当分支预测错误时，它浪费的周期数比 RISC 处理器要大。在超标量体系结构中，每个周期要求读取多条连续的指令，ISA 的复杂性加剧了取指硬件的复杂性，特别是动态指令序列中含有分支和跳转指令时，不过 trace cache 机制可以解决大部分的问题。

3.5 超长指令字微结构

动态调度流水线有很多优点，但是也有一些缺点，这些缺点限制了我们难以获得更宽的分发带宽。由于动态微结构试图每周期分发更多的指令，因此硬件结构也变得更加复杂、速度更慢，也更耗电。从某种程度上讲，乱序执行结构是一个极端，因为所有的决策都是硬件在执行时动态做出的。

我们将会在本部分介绍另一个极端情况：所有决策（包括取指、分发、调度）所用的微

结构都在编译时静态实现的。

3.5.1 动态和静态技术

大多数动态乱序执行微结构中采用的执行技术都可以在编译器中以某种方式实现，主要的区别在于：(1) 相比硬件执行时编译器获得的可靠信息更少；(2) 编译器具有一些硬件没有的代码高级视图。

在动态乱序执行机器中，程序员强制代码以数据流方式进行乱序调度，而不是按照线程序顺序执行。指令允许相互跳过，甚至当某些指令由于操作数没准备好而被阻塞时，线程序中后面的指令也可以开始执行。这种情况下必须解决数据冲突和控制冲突（条件分支和异常），而解决方法包括动态寄存器重命名、动态内存歧义消除、分支预测、基于重排序缓冲区和回退机制的推测执行机制等。

在静态机器中，编译器首先根据程序员的意图按序编译源代码，然后编译器试图对代码进行适当移动以提高性能。由于对指令的执行顺序进行了重新组织，这种代码移动实际上是另一种乱序执行指令调度方式。代码移动可以是局部的（例如，原始 Tomasulo 算法只在每个基本块内移动）或者是全局的（例如，推测执行可以跨越基本块）。全局调度通常根据调度是否适用于循环结构划分为循环调度和非循环调度。循环调度技术包括循环展开和软件流水。动态体系结构也可以采用循环展开，它通过分支预测和在重排序缓冲区调度循环结构中不同迭代的指令来实现。另一方面，软件流水只用于静态调度，因为它对代码结构做了特定假设。非循环调度应用于非循环代码，例如踪迹调度。在踪迹调度中，编译器首先基于静态分支预测的最可能的程序 trace 进行指令发射。之后，编译器依次根据所有其他可能的 trace 发射指令，并且使用补偿代码消除每个 trace 中可能产生的副作用。在动态推测执行机器中，类似的方式可以通过预测分支执行相应的 trace，当碰到分支预测错误时进行回退这样的机制来完成。

对于静态分支预测，编译器利用所有能够收集到的信息来判断每次分支的方向，这一工作通过剖析代码来实现。获得的信息之后通过附加到分支操作码上的一个提示位传递给硬件。编译器也可以对寄存器进行重命名，但是它只能对结构寄存器重命名，这类寄存器的数量受到 ISA 的限制。由于内存地址在编译时是未知的，静态内存歧义消除很难实现。此外将 load 指令移到不同地址寄存器的 store 指令之前也是非常危险的，因为编译时未获知寄存器的值。与动态体系结构类似，当我们推测将 load 指令移至 store 指令之前时，需要有一些特殊机制保障，当动态检测到相关（或冲突）时，可以用补丁代码进行恢复。

最后，必须重视异常模型。动态体系结构中，可以通过回退运行机制处理检测到的异常。而在静态体系结构中，运行基本块边界外的指令可能导致不应出现的异常。我们需要特定的机制可以保证只处理想要的异常而忽视掉任何不应出现的异常。当不应出现的异常出现时，必须部署诸如补丁代码这样的恢复机制来取消不应出现的异常。

在 LIW/VLIW 体系结构中，从开始取指到指令运行结束，编译器处理指令执行中的所有步骤。它可以最小化硬件复杂度和功耗，给超标量体系结构提供很大的取指和分发带宽，提供动态调度机器所无法想象的优势。

3.5.2 VLIW 体系结构

超长指令字适用于静态调度微架构，它的每条长指令包含几个叫作 op 的 MIPS 命令。程序计数器指向一条长指令，一次取出长指令中的所有 op。这些 op 随后被译码、执行、写回。由于每个 op 操作作用于不同的流水线，因此其编码可能不同，并且可以针对每条特定流水线进行优化。编译器解决了所有的冲突，它通过在相关的源（父）指令和目的（子）指令之间插

入足够多的指令（执行时对应很多个时钟周期）解决了寄存器 RAW 冲突，通过重命名寄存器解决了 WAW 和 WAR 冲突，通过合适的代码调度避免了分支和跳转指令导致的结构和控制冲突，通过添加补丁代码解决了异常和内存数据访问导致的冲突。

图 3-31 给出了一种带有 5 个 op 槽的 VLIW 机器：两个 load/store 槽、两个浮点槽、一个整型和分支槽。每一个 op 槽可能是 16 ~ 32 位，整个长指令长度为 80 ~ 160 位。

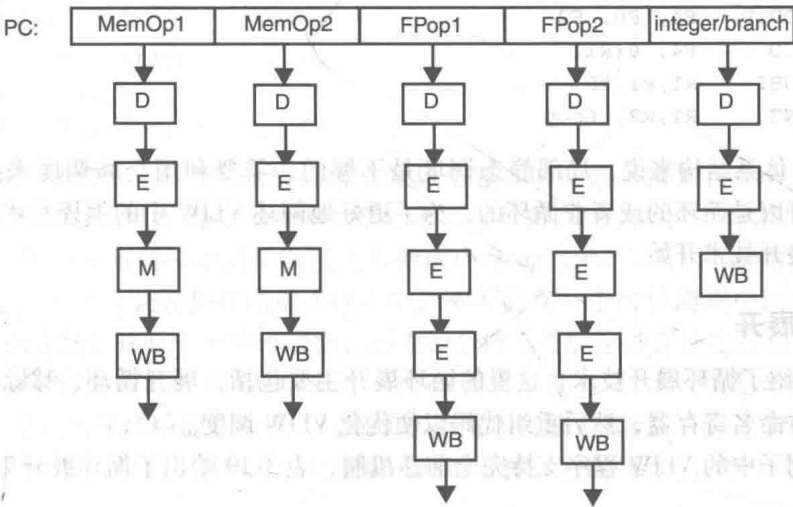


图 3-31 五路 VLIW 机器

这种微结构除了用于激活执行流水级的控制点之外，只需要非常少的硬件控制逻辑，尤其是不需要冲突检测单元。前递技术有一定作用，但不是必要的。长指令在取指、译码和执行时，不存在任何数据冲突导致的阻塞。如果没有前递机制，那么相关指令在其父指令还没有完成流水线中写回阶段之前不能进行静态调度。如果支持寄存器前递机制，在值写入寄存器的同时，译码阶段的操作数就可用了，因此，相关指令在其父指令进入了写回阶段时就可以进行静态调度。而如果支持完全前递（full forwarding）机制，那么指令只要在其父指令的结果出来时就可以静态调度。

表 3-18 给出了图 3-31 所示架构在不同前递假设下的操作时延，编译器在优化时只需要知道这些时延即可。更多的前递意味着更少的操作时延，因此可以从给定指令级并行度的负载中获得更紧凑的代码和更快的加速比。

表 3-18 不同前递假设下的操作时延

源	目的	无前递支持	支持寄存器前递	支持完全前递
Load	any	3	2	1
Integer	any	2	1	0
FP ALU	any	4	3	2
Store	load	0	0	0

因为每个译码器专门用于特定指令类，所以译码器可以很简化，速度更快。此外，不同位置 op 槽的格式（包括大小）也可以不同。在这举例的机器中，分支被延迟两条指令来避免刷流水线。这是从硬件角度来看最简单的解决方案。

为了提高效率，这种简单的硬件结构需要有效的编译器支持，将 op 打包为长指令，使得它们可以在没有硬件控制的情况下进行取指、译码、分发和执行。我们将在本节通过对下面的

代码例子进行详细分析来阐述编译算法。

```
FOR (i = 1000; i > 0; i = i-1)
    x[i] = x[i] + s
```

上述代码编译成如下指令序列：

```
Loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        SUBI    R1, R1, #8
        BNE     R1, R2, Loop
```

对于 VLIW 体系结构来说，局部静态调度是不够的，需要利用全局调度来挖掘指令级并行。全局调度可以是循环的或者非循环的，为了更好地阐述 VLIW 中的编译技术，我们从循环调度中的循环展开技术开始。

3.5.3 循环展开

3.3.5 节介绍了循环展开技术，这里的循环展开主要包括：展开循环，移除分支指令，调整内存偏移，重命名寄存器，然后重组代码以便优化 VLIW 调度。

假设上面例子中的 VLIW 程序支持完全前递机制，表 3-19 给出了循环展开 7 次的情况。

表 3-19 循环展开 7 次后的 VLIW 程序

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Integer/Branch
1	L D F0, 0(R1)	L D F6, -8(R1)	NOOP	NOOP	NOOP
2	L D F10, -16(R1)	L D F14, -24(R1)	NOOP	NOOP	NOOP
3	L D F18, -32(R1)	L D F22, -40(R1)	ADD. D F4, F0, F2	ADD. D F8, F6, F2	NOOP
4	L D F26, -48(R1)	NOOP	ADD. D F12, F10, F2	ADD. D F16, F14, F2	NOOP
5	NOOP	NOOP	ADD. D F20, F18, F2	ADD. D F24, F22, F2	NOOP
6	S. D 0 (R1), F4	S. D -8(R1), F8	ADD. D F28, F26, F2	NOOP	SUBI R1, R1, #56
7	S. D 40(R1), F12	S. D 32(R1), F16	NOOP	NOOP	DBNE R1, R2, CLCK1
8	S. D 24(R1), F20	S. D 16(R1), F24	NOOP	NOOP	NOOP
9	S. D 8(R1), F28	NOOP	NOOP	NOOP	NOOP

每条超长指令包含 5 个 op，对应图 3-31 中的 5 条流水线。根据 op 的操作码将 op 放置在每个指令槽。指令排布时必须考虑寄存器操作数的操作时延。例如，在 load 指令和相关的 add 指令之间必须插入一条长指令，在 add 指令和相关的 store 指令之间必须插入两条长指令。因为 subtract 指令是零延迟，分支指令可以在 subtract 指令之后立即调度。在所有情况下，分支指令将延迟两个时钟周期以避免刷流水线。

该 VLIW 程序展现了循环展开的一些缺点。首先，由于循环体被重复执行多次，代码会显著膨胀。第二，由于 op 调度时必须避免寄存器上的数据冲突，因此导致大量的发射槽被浪费（填充 NOOP 指令）。第三，寄存器压力太大，结构寄存器的总数限制了循环展开的次数。

3.5.4 软件流水

软件流水技术会对原始循环不同迭代中的指令进行重新组织，以便重组后指令可以在新的流水循环的一次迭代中执行。为了阐述基于 VLIW 背景下的软件流水，以如下核心代码作为运行实例。该核心代码中的每条指令将被编码成 VLIW 指令中的 op，将其分别标记为 O1，O2，O3：

```
Loop:  L.D    F0, 0(R1)      O1
        ADD.D  F4, F0, F2    O2
        S.D    F4, 0(R1)    O3
```

表 3-20 给出了原始循环中 7 次迭代的流水循环是怎么调度的。调度表中的每一列是原始循环中的一次迭代，为了满足每个 op 时延的要求，有依赖关系的 op 之间都用足够的距离（行）分隔开了。因为原始循环中的每次迭代有两次内存 op 操作（一次 load 操作和一次 store 操作），所以在只支持两个访存操作槽的 VLIW 中，不可能在一个时钟周期中同时调度初始循环中的一个以上的迭代。这是一个物理限制，因为每个槽不可能超过百分之百利用。表中的每一行对应最终 VLIW 程序中每条 VLIW 指令的一些 op 操作。注意 INST1 ~ 5 是软件流水的前序部分；INST6 ~ 7 是流水循环中的两次迭代（INST6 是循环的核心代码）；最后 INST8 ~ 12 组成收尾部分。这里面，核心代码（INST6）是流水循环的主体，它的每个 op 操作将放置到 VLIW 的一个指令槽中。

表 3-20 基于软件流水的调度表

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6	ITE7
INST1	O1						
INST2	—	O1					
INST3	O2	—	O1				
INST4	—	O2	—	O1			
INST5	—	—	O2	—	O1		
INST6	O3	—	—	O2	—	O1	
INST7		O3	—	—	O2	—	O1
INST8			O3	—	—	O2	
INST9				O3			O2
INST10					O3		
INST11						O3	
INST12							O3

每个核心代码的迭代都由一条 VLIW 指令执行。根据调度，核心代码中每条 VLIW 指令的 store 指令比 load 指令晚 5 个迭代。因此，store 指令的地址偏移必须进行调整。表 3-21 给出了核心代码的运行结果。

表 3-21 软件流水的 VLIW 核心代码

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Integer/Branch
1	L. D F0, 0(R1)	S. D 40 (R1), F4	ADD. D F4, F0, F2	NOOP	

在表 3-21 的核心代码中，忽略了循环控制 op（例如，延迟分支和地址寄存器的更新），这些操作一般分配到整型/分支槽，这样做是为了简化例子。注意这里的核心代码非常短（一条

指令)，而实际（复杂）情况是，核心代码可能需要三条以上的 op 操作以及三条以上的指令，以便使整型/分支流水线中有足够的操作槽用于调度循环控制指令。

使用旋转寄存器解决 WAR 冲突

我们已经介绍了怎样通过调整位置解决寄存器中的 RAW 冲突和所有内存中的冲突。WAW 冲突不存在，因为目的寄存器是按顺序更新的，并且每条指令的 op 操作按时钟级同步。如果没有异步事件中断调度，WAR 冲突也可以避免。如果没有事件扰乱表 3-20 中按时钟周期顺序的调度，那么即使相同寄存器的多个值在同时计算，所有寄存器的值也都能立即获得并且从源指令前递到目的指令。

例如，表 3-21 中 FPOp1 槽中的 ADD.D 指令更新 F4 寄存器后，该寄存器的值必须作为三个时钟周期后的 store 指令的输入操作数。每个时钟周期都会启动一条修改 F4 寄存器值的新 ADD.D 指令，这样 F4 寄存器同时有多个值在计算。不管怎样，只要处理器没有停下来，F4 寄存器的值会被立刻从浮点流水线前递到访存流水线（从 add 操作到 store 操作）。然而，如果因为某个原因，处理器在这两个 op 操作之间停下来（例如，碰到 cache miss 或者异常），那么 F4 寄存器的值可能在 store 操作之前被随后的 add 操作更新，并且可能跟着发生 WAR 冲突。再者，一个寄存器的值一般可以在不同时钟周期作为两个不同可调度 op 操作的输入。这种情况下，为了避免 WAR 冲突，这两个 op 操作都必须以最坏情况下的时延来调度。寄存器重命名可以帮助避免这些冲突，并有助于生成最有可能的调度。

在 VLIW 体系结构中，旋转寄存器是一种解决 WAR 冲突的常用硬件方案。旋转寄存器堆是一个特殊的寄存器堆，它在软件流水循环中自动将结构寄存器重命名为物理寄存器。

图 3-32 从硬件层说明了旋转寄存器的概念。循环的每次迭代时，旋转寄存器 $i(RR_i)$ 会被映射为不同的物理寄存器 $j(P_j)$ 。从旋转寄存器到物理寄存器号的映射由旋转基址寄存器 (RRB) 的内容控制。每当控制循环的分支指令执行时，RRB 就加 1。RRB 寄存器加上 RR 号，再与寄存器堆大小求模，这样就可以获得 ISA 访问的旋转寄存器所对应的物理寄存器号。

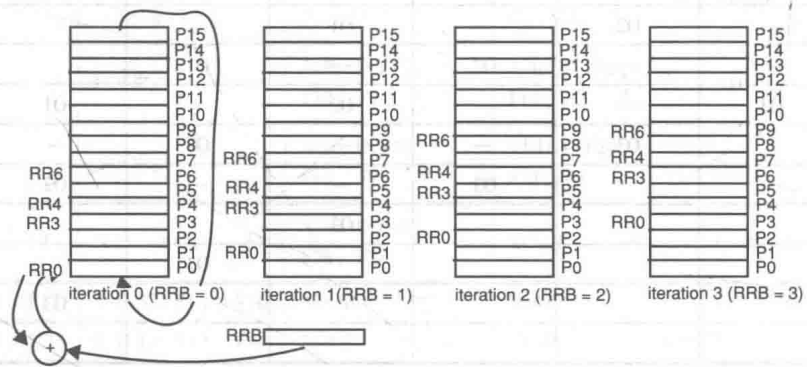


图 3-32 旋转寄存器

旋转寄存器堆在软件流水循环阶段自动解决寄存器 WAR 冲突。保存常量的寄存器被分配到常规的整型或者浮点寄存器。保存循环迭代时访问变量的寄存器被分配给旋转寄存器。回到刚才的例子，F2 寄存器保存常量，F0 和 F4 寄存器则分配给旋转寄存器。例如，O3 行将 RR0 寄存器分配给 store 指令的源寄存器。在第 0 次迭代时，RR0 寄存器映射到 P0 寄存器。RR0 寄存器必须保存三个迭代之前由 add 指令 (O2) 生成的用于 store 的值。因此，编译器将 O2 的目的寄存器分配给 RR3，RR3 寄存器在第 0 次迭代时被映射为 P3，进而在第 3 次迭代时（三个时钟周期之后），RR0 寄存器被映射为 P3。然后编译器将 O2 的第二个源寄存器分配给 RR4，RR4 在第 0 次迭代时被映射为 P4。第二个源寄存器是 load 指令的运行结果 (O1)，它分配给

RR6, RR6 在第 0 次迭代时被映射为 P6, 以便两次迭代之后, 在时钟周期 2 时, RR4 能够映射到 P6, 并且 O2 行的指令能够读取到两个迭代之前的 O1 行指令的运行结果。图 3-32 显示了循环的 4 个迭代 (迭代 0、1、2、3) 中 4 次连续的旋转寄存器到物理寄存器的映射。在迭代 0 时, RR_i 映射到 P_i 。在迭代 1 时, RR_i 映射到 $P_j, j = (i + 1) \bmod 16$ 。

表 3-22 给出了 VLIW 核心代码的结果。循环核心代码可以在一个时钟周期执行, 并且解决了寄存器和内存值的所有冲突。前序和收尾部分的程序变得更复杂, 但是它们只需要执行一次。

表 3-22 支持旋转寄存器的软件流水 VLIW 核心代码

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Integer/Branch
1	L D RR6, 0(R1)	S D 40(R1), RR0	ADD D RR3, RR4, F2	NOOP	

Op 槽冲突

可获得的 slot 槽数量限制了指令槽上的 op 操作的分配。假设 VLIW 机器仅仅含有一个内存 op 槽位和一个浮点 op 槽位。因为 loop 循环包含两次内存 op 操作, 不可能在一个时钟周期中调度多个 loop 迭代。表 3-23 展示了调度过程, 其中 INST1 ~ 4 构成程序的前序部分, INST9 ~ 12 构成程序的收尾部分, INST5 ~ 6 构成了循环核心。调整完内存地址偏移并且完成旋转寄存器赋值之后的 VLIW 程序, 如表 3-24 所示。为了简化起见, 再次忽略 loop 控制 op 操作。

表 3-23 带 slot 限制的软件流水调度表

	ITE1	ITE2	ITE3	ITE4
INST1	O1			
INST2	—			
INST3	O2	O1		
INST4	—	—		
INST5	—	O2	O1	
INST6	O3	—	—	
INST7		—	O2	O1
INST8		O3	—	—
INST9			—	O2
INST10			O3	—
INST11				—
INST12				O3

表 3-24 仅有一个内存单元、支持旋转寄存器的 VLIW 核心代码

Clock	MemOp	FPOp	Integer/Branch
1	L D RR3, 0(R1)	ADD D RR1, RR2, F2	
2	S D 24(R1), RR0	NOOP	

一般而言, 每种指令类型都对应有限个可分配的操作槽数量。令 K_i 表示 i 类型指令的操作槽数量, N_i 表示 loop 循环体中 i 类型指令的数量。由于所有指令类型产生的操作槽冲突, VLIW 指令数量的最小值可以根据下面的公式进行计算:

$$\text{MAX}_i \left\lceil \frac{N_i}{K_i} \right\rceil$$

跨循环依赖

除了可用 op 槽数量的影响外，指令级并行（ILP）程度也决定了 VLIW 核心代码的长度。软件流水的目标是降低每个循环迭代内的相关性（依赖），但是也受限于跨循环的相关。跨循环依赖是在一个循环不同迭代的指令之间的一种相关性，这种相关性非常普遍（例如在递归中），并且会限制循环中指令级并行（ILP）的程度。

考虑如下这个具有跨循环依赖的代码：

```
for (i = 0; i < N; i++)
{
    A[i+2] = A[i]+1;
    B[i] = A[i+2]+1;
}
```

第一条语句中 A[i] 的相关有两个迭代的距离，相应的 MIPS 代码（假设是单精度浮点格式）及其数据依赖图（Data Dependency Graph, DDG）如图 3-33 所示。

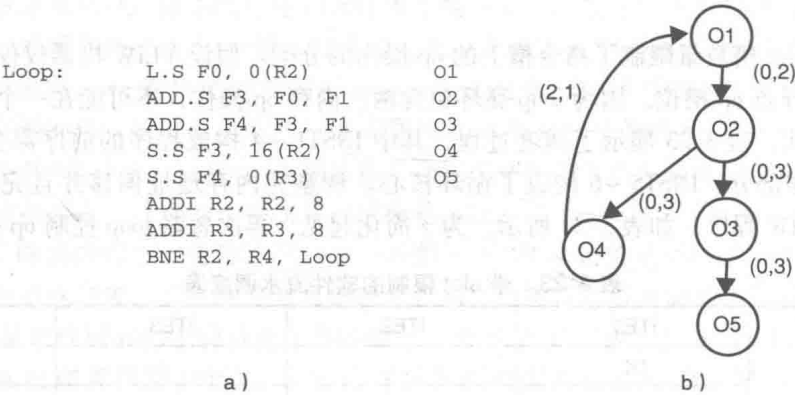


图 3-33 跨循环迭代相关的代码 (a) 及其数据依赖图 (b)

DDG 图中的每个节点是循环体中的一条指令，每条边代表两条指令（或者 VLIW 中的 op 操作）之间的相关性。边用一个数对 (diff, min) 进行标记，其中 diff 代表两条指令之间的迭代数，min 代表为了避免 RAW 冲突两条指令之间的最小周期数。跨循环依赖导致了图中出现环，DDG 图中的环限制了 loop 迭代调度的速率。例如，图 3-33 中的环跨越两次迭代，在这个环中为了避免 RAW 冲突所需的最小时钟周期是 6。因此，如果调度速度快于每 3 个周期调度一次循环迭代，将导致不安全结果出现。

每 3 个时钟周期调度一个迭代的过程如表 3-25 所示。INST1 ~ 6 是程序的前序部分，INST13 ~ 15 是程序的收尾部分，INST7 ~ 9 构成了流水循环的核心。

表 3-25 图 3-33 中的循环调度

	ITE1	ITE2	ITE3	ITE4
INST1	O1			
INST2	—			
INST3	O2			
INST4	—	O1		
INST5	—	—		
INST6	O3, O4	O2		
INST7	—	—	O1	
INST8	—	—	—	
INST9	O5	O3, O4	O2	

(续)

	ITE1	ITE2	ITE3	ITE4
INST10		—	—	01
INST11		—	—	—
INST12		05	03, 04	02
INST13			—	—
INST14			—	—
INST15			05	03, 04

根据调度表将 op 操作分配给指令槽之后，再修改内存地址偏移，分配旋转寄存器。支持两个内存槽、两个 FP 槽、一个整型/分支槽的 VLIW 核心代码如表 3-26 所示。

表 3-26 支持旋转寄存器的软件流水 VLIW 核心代码

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Integer/Branch
1	NOOP	L S F4, 0(R2)	NOOP	NOOP	DBNE R2, R4, CLK1
2	NOOP	NOOP	NOOP	NOOP	ADDI R2, R2, #8
3	S. S RR2, 0(R2)	S. S RR0, -16(R3)	ADD. D RR3, F4, F1	ADD. D RR1, RR2, F1	ADDI R3, R3, #8

注意，没有必要将旋转寄存器分配给 load 指令的目的寄存器，因为 load 值是被同一个迭代中的 add 指令使用的。因此，给它分配了一个普通的 FP 寄存器（F4 寄存器）。最后，由于 VLIW 程序包含三条指令，我们在表 3-26 中添加了循环控制 op 操作，这类 op 在整型/分支流水线中执行。分支（DBNE）延迟两条指令，当发生跳转时，将 RRB 值加一。

软件流水算法

要实现软件流水，编译器必须识别循环中的 op 操作，然后构造 DDG 图。图中的每一个节点是循环体中的一个 op 操作，每条边反映两个 op 操作的相关性，用一个数对（diff，min）标记，其中 diff 是迭代数量，min 是两个相关 op 操作之间想要避免相关所需的最小时钟周期数。

编译器必须找出图中所有的环。对于图中每个环 C_k ，编译器做如下处理：

- 通过计算环 C_k 所有边的 min 之和（叫作 $MIN(C_k)$ ），来获得环所需的最小时钟周期数；
- 通过计算环 C_k 所有边的 diff 之和（叫作 $DIFF(C_k)$ ），来获得环所跨越的迭代数；
- 计算 $\Pi(C_k) = MIN(C_k)/DIFF(C_k)$ ，向上取整。

所有环 C_k 对应的所有 $\Pi(C_k)$ 中的最大值就是由于跨循环依赖造成的两个连续迭代之间所需的最小时钟周期数（表示为 Π_1 ）。

然后编译器还必须考虑 op 槽的冲突，对于每种类型的 op O_k ，编译器做如下处理：

- 计算该循环每个迭代中类型 O_k 的 op 操作的数量（叫作 $NUM(O_k)$ ）；
- 根据类型 O_k 对应的 op 槽数（记为 $SLOTS(O_k)$ ），计算 $\Pi(O_k) = NUM(O_k)/SLOTS(O_k)$ ，向上取整。

所有 op 操作类型所对应的 $\Pi(O_k)$ 中的最大值就是由于 op 槽冲突造成的两个连续迭代之间所需的最小时钟周期数（表示为 Π_2 ）。

两个连续迭代之间的最小时钟周期数就是 Π_1 和 Π_2 之间的最大值，我们把这个值叫作 $MIM\Pi$ 。

然后，编译器在两个连续迭代之间基于一个固定长度时钟周期（ $\Pi = MIN\Pi$ ）进行软件流水循环的调度，并且试图找出一个重复的计算核心。如果没有找到，则用 $\Pi + 1$ 的时间间隔重复这个调度过程，直到找到重复计算核心为止。

一旦发现可重复的核心代码调度方式，编译器就开始产生前序、核心、收尾部分的代码。具体步骤包括：根据调度分配指令槽给 op 操作，调整地址偏移，将旋转寄存器的值赋给变量寄存器操作数等。

3.5.5 非循环 VLIW 调度

虽然循环结构在科学和工程计算中占据最大的计算量，但是根据 Amdahl 定律，要想获得好的整体性能，其他代码也必须高效执行。而且，循环调度仅仅是当循环体中没有条件分支时才起作用。因此，除了处理 loop 循环，编译器也必须处理包含条件分支的普通代码。一种用于这类代码调度的技术叫作 trace 调度，属于非循环调度的一种。

trace 调度从静态分支预测开始。通过分析代码，编译器可以决定哪些分支是可以静态预测的，如果一个分支可以静态预测，那么 trace 调度就有效。

图 3-34a 展现了含有条件分支的简单代码例子，代码中（典型的“if-then-else”结构）A、B、C、D 是代码块。如果分支测试 b 的值是静态可预测的，编译器首先假设将从最可能的路径执行。

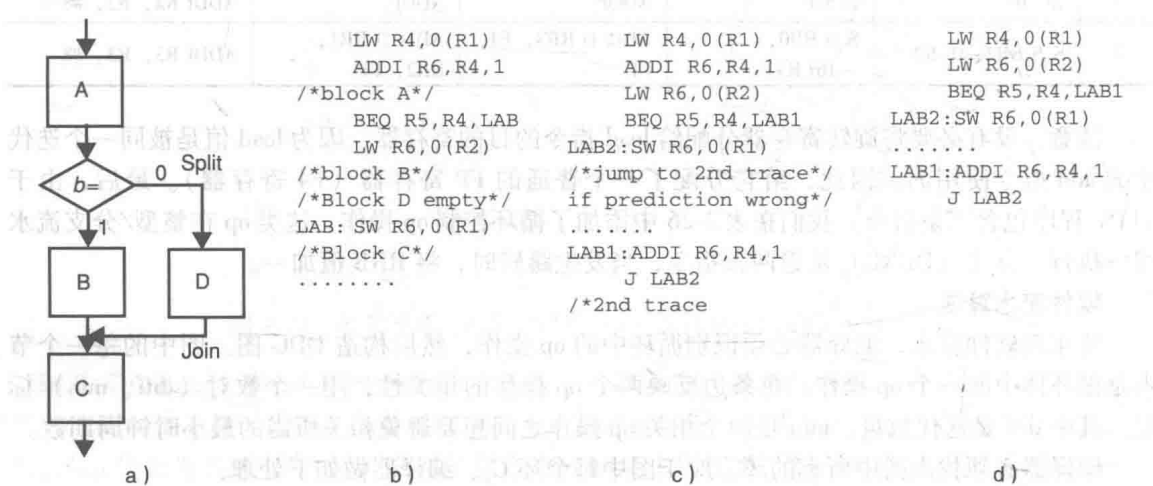


图 3-34 含有 if-then-else 语句的代码

假设这条路径是块 A，然后是块 B 和块 C（分支没执行）。编译器先将这个 trace 看成是最可能执行的 trace，并首先进行调度，就好像没有分支一样。编译器通过移动代码的方式来避免数据冲突，但是忽略分支产生的控制冲突。在执行时，一旦知道了分支结果，就可以确定该预测是否是正确的。如果预测正确，则继续执行，从而避免了分支引起的控制冲突。尽管如此，编译器也必须考虑分支预测错误的情形。因此，编译器必须调度另一个 trace（该 trace 执行指令块 D）。此外，编译器必须向第二个 trace 中添加一些补充代码。补充代码用来修复更新块 B 和 C 所造成的影响，因为该更新可能影响块 D 中的计算（对应图中 join），也还用来修复更新块 B 和 D 所造成的影响，因为块 B 和块 D 的更新可能影响块 C 的结果（对应图中 split）。因此，编译器用块 D 代码加上补充代码（split 和 join）调度第二个 trace，第二个 trace 仅当分支预测错误时才执行。

如果分支结果的静态预测是可靠的，trace 调度程序大多数时候只运行第一个 trace，就好像没有条件分支似的。图 3-34a 中的每个独立块可能也包含一些条件分支指令，这意味着可能生成许多 trace。

图 3-34b 展示了一种可能的代码，其在块 D 中是空的。图 3-34c 展示了 trace 调度之后分支

静态预测不跳转时的代码。块 B 中的 load 指令在第一个 trace 里已经被移到了分支之前。通常来讲，将 store 指令移到分支的前面（也在 split 前）是不安全的也是不必要的（因为 store 操作不能取消），store 指令更应该往后移。如果分支预测错误，执行第二个 trace 的话，因为块 D 是空的，第二个 trace 只包含一些补充代码（split 和 join）。图 3-34d 展示了对第一个 trace 进一步优化后的代码，已经将第一个 trace 中的 ADDI 指令移除，下一步应当在目标机器上调度 VLIW 程序。如果静态分支预测正确，唯一运行的代码是两条 load 指令、一条分支指令和一条 store 指令。优点之一是两条 load 指令现在可以并行发射。

尽管从数据流的观点来看，图 3-34d 和图 3-34b 的运行结果一样，但是它们对应的异常行为可能是不同的。前移的 load 指令现在是推测执行的，但是无论分支是否跳转，load 总是在第一个 trace 中执行。如果当前移的 load 造成一次异常并且分支跳转成功，实际上该异常是不应该发生的，此时需要一些额外的支持来处理这种异常。

3.5.6 谓词指令

当分支结构是倾向性并且静态高度可预测时，trace 调度就非常有效，因为最可能的 trace 是运行时间最多的一个。然而，一些条件分支很难静态预测或者没有明显倾向性。对于这些分支来说，分支 trace 调度表现得很糟糕，因为有多个 trace 会被频繁执行。

为了高效执行带有无倾向性分支的代码，可以利用 VLIW 中丰富的并行性。如果提供了谓词执行，无倾向性分支的两个方向都可以并行执行。谓词指令（或者叫条件指令）仅当存储在谓词寄存器中的谓词为真时才执行，否则，指令或者指令结果就被丢掉。表 3-3 中的任何指令都可以是谓词指令。例如，可以将 LW 的两个有条件指令版本加入到 MIPS 指令集：

```
CLWZ R1,0(R2),R3      /*Load Mem[0(R2)] into R1 if R3 is 0
CLWNZ R1,0(R2),R3     /*Load Mem[0(R2)] into R1 if R3 is not 0
```

这里，R3 是谓词寄存器，除非谓词寄存器中的值满足条件，否则不允许条件指令修改体系结构的状态或者抛出异常。尽管谓词指令的长度和格式与非谓词指令（需要增加一个寄存器操作数）非常不同，但是在 VLIW 中这不是问题，VLIW 并不需要每种 op 类型都遵守统一的格式。条件指令将控制相关（不利于代码移动）转化为寄存器中的数据相关（编译器很容易处理这种相关）。

为了说明 VLIW 中谓词指令（条件指令）的用法，再考虑图 3-35a 中的简单例子，假设 BEQ 无法静态预测或者分支无倾向性。图 3-35b 中，分支已经去掉，load 指令现在成了一个条件 load 指令。图 3-35c 展示了代码进一步优化后的版本，其中两条条件指令可以并行执行。尽管这个指令序列由于相关无法进一步优化，但是分支的移除还是给编译器提供了更多的机会进行代码排布。

LW R4,0(R1)	LW R4,0(R1)	LW R4,0(R1)
ADDI R6,R4,#1	ADDI R6,R4,#1	SUB R3,R5,R4
BEQ R5,R4,LAB	SUB R3,R5,R4	CADDIZ R6,R4,#1,R3
LW R6,0(R2)	CLWNZ R6,0(R2),R3	CLWNZ R6,0(R2),R3
LAB: SW R6,0(R1)	SW R6,0(R1)	SW R6,0(R1)
.....
a)	b)	c)

图 3-35 谓词指令的使用

一般而言，当分支不可预测且被分支所对应的基本块较小时，条件指令就非常有用。对于 VLIW 之外的其他微结构，例如乱序执行动态处理器，谓词指令也是有用的，它可以用来避免小基本块中分支预测错误所产生的较高开销。不过，谓词指令技术会增加后端（尤其是调度

器)的复杂性。某些 ISA (例如, Intel 的 IA-64 架构) 中, 所有指令都是谓词指令。

3.5.7 推测内存歧义消除

分支指令是编译器进行代码移动的一个障碍, 此外, 内存歧义消除是另一个障碍。也就是说, 如果一条 load 指令和一条 store 指令在编译时不能断定它们的内存地址一定不同, 那么这条 load 指令就不能移动到 store 指令之前。然而, store/load 指令序列的内存地址大多数时候都是不同的。因此, 编译器可以试探着将一条 load 指令移到 store 指令之前, 即使这两个地址在编译时还无法确定是否相同。为了检测地址冲突, 需要在 load 指令的原来位置插入一条特殊指令——监护指令 (guardian)。监护指令检测是否有 store 指令修改了之前推测执行 load 指令所访问的内存地址, 如果有, 那么出现了 RAW 违例, 需要执行对应的修复代码。

图 3-36a 中, 编译器无法将 load 指令移到 store 之前, 因为寄存器 R2 和 R3 的内容在编译时是未知的。图中的 I1 和 I2 是独立不相关的指令。在图 3-36b 中, load 指令和一条依赖 load 的指令被推测前移到 store 指令之前。这里的代码移动可能导致出现内存 RAW 冲突, 因此必须在运行时进行验证。在图 3-36c 中, 在 load 指令的初始位置加入了一条监护指令 (检测指令), 而 load 指令则被标记为推测的 (LW. a)。检测指令动态验证 load 指令操作的内存位置有没有被 store 指令修改, 如果被修改过, 则调用 repair 处理程序进行修复。为了检测冲突, 推测的 load 指令会将它访问的内存地址插入一个短小的硬件全相联表中, 然后后续的每条 store 指令会查询这个表中的内存地址, 并且将与自身匹配的地址从中移除。检测指令会检查 load 指令的地址是否仍然在这个内存表中, 如果不在, 则调用修复处理程序。图 3-36c 中, 推测指令 LW. a 将地址 0 (R2) 插入到硬件表中, 后面的 store 指令 SW 通过地址 0 (R3) 查询硬件表, 如果地址在表中, 则将其移除。之后, 指令 CHECK. a 查询硬件表, 如果指令没有在硬件表中找到地址 0 (R2), CHECK. a 指令跳到修复程序, 修复程序将非推测地重新执行这两条指令 (LW 和 ADD)。

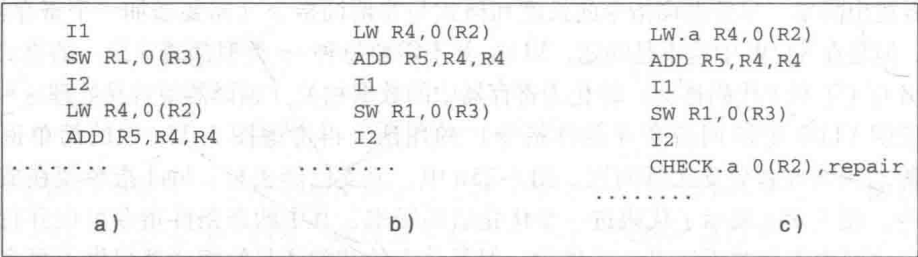


图 3-36 使用监护指令消除推测内存歧义

3.5.8 异常

VLIW 体系结构不擅长处理诸如 cache miss 和异常这样的动态事件。当某个 op 操作发生了 cache miss, 整条指令都必须停止, 因为编译器产生的静态调度依赖于指令每步有序地执行和在相同长度指令字中的 op 操作按部就班地同步执行。因此, 当出现异常时, 整条流水线可能被完全冻结, 就像在静态流水线中见到的一样。通过冻结整条流水线, 可以保证编译器所要的精确调度方式保持不变。

在 VLIW 结构中, 不管是在硬件还是软件层面上, 异常都更难以处理。在硬件层级中, 没有像动态乱序执行流水线那样的保存在重排序缓冲区中的历史运行记录, 因此, 执行过程无法精确回滚到错误指令处。VLIW 体系结构的主要优点就是硬件相当简单。在软件层级, 为了优化代码的执行时间, 编译器会推测地将一些指令提到条件分支前面。对于循环调度来说这不是

个问题，因为所有在编译后循环（展开后循环或者流水循环）中执行的指令也同样会在原始循环（程序员本意的循环）中执行。然而，对于全局的非循环调度来说这是个问题，因为每个 trace 中有许多指令是被编译器推测调度执行的，在运行时可能触发一些不应出现的异常。

通常，有些异常（诸如算术溢出或者地址越界等）会影响用户程序的执行，从这种意义上说，它们对用户是可见的。而其他异常（例如缺页）则对用户不可见，它们是透明的。用户不可见异常通常没有什么副作用，因此即使这种异常是本不应该出现的，也可以直接处理。然而，用户可见异常就不同了，如果它们本不应该出现的话就必须进行限制。

VLIW 结构中，如果异常（即使是本不应该出现的）对用户不可见，那么通常会直接处理，因为这种异常要么是确实需要的，要么是没有副作用的。然而，像被编译器推测移到分支前面的指令所产生的用户可见的异常，是绝不能直接处理的，除非推测正确。例如，一条 load 指令在编译的时候被移到了条件分支前面，由于它对于程序的正确执行没有影响，硬件可以并且也会处理 load 产生的缺页异常。然而，如果相同的推测执行 load 引起一次终止异常，例如出现地址未对齐异常或者非法访问异常，那么这种异常只有在这次推测执行的 load 实际上也应当被执行时才会被处理。

图 3-37a 中给出了来自图 3-35a 的原始代码，trace 调度和优化后的代码结果显示在图 3-37b 中。推测的 load 指令在第一个 trace 时被移到分支前面可能触发缺页异常，这不会影响正确性，且总是被直接处理。然而，如果该前移的 load 操作触发了一次终止异常，并且分支指令跳转的话，那么程序也将终止运行，这显然是程序员所不期望的结果。因此，如果我们忽略用户可见的异常的话，那么程序员就没法察觉到推测代码的错误。不幸的是，大多数 ISA 的异常模型都不允许禁止用户可见异常（包括终止异常），并且禁止用户可见异常的做法无法避免程序员写出错误程序。在禁止用户可见异常的情况下，程序执行中可能发生一个未被检测到的程序错误，这个错误可能会长期甚至一直都没有被检测到，直到程序运行崩溃。

LW R4,0(R1)	LW R4,0(R1)	LW R4,0(R1)	LW R4,0(R1)
ADDI R6,R4,1	LW R6,0(R2)	sLW R6,0(R2)	LW.s R6,0(R2)
BEQ R5,R4,LAB	BEQ R5,R4,LAB1	BEQ R5,R4,LAB1	BEQ R5,R4,LAB1
LW R6,0(R2)	LAB2:SW R6,0(R1)	LAB2:SW R6,0(R1)	CHECK.s R6,repair
LAB: SW R6,0(R1)	LAB2:SW R6,0(R1)
.....	LAB1:ADDI R6,R4,1	LAB1:ADDI R6,R4,1
	J LAB2	J LAB2	LAB1:ADDI R6,R4,1
			J LAB2
a)	b)	c)	d)

图 3-37 异常支持

为了解决当用户可见异常被禁止时程序失控造成的问题，异常可能被延迟。当编译器将一条指令移到条件分支前面时，会将其标记为推测执行。通常，除了 store 指令（不能推测执行）之外的所有指令都可能推测执行。用户不可见的异常仍然可以在引发异常的指令处被直接处理，但是对于用户可见的异常，只有当发生在非推测执行指令上时才被立即处理。当推测执行的指令触发了用户可见异常时，该异常会被延迟。我们将异常指令的目的寄存器标记为“有毒的”（poisoned）并且填入一个伪造值或者一个异常报告，当这个有毒值被其他推测指令读取时，就会传到读指令对应的目的寄存器中，并且不产生异常。当有毒值被某条非推测执行指令作为输入操作数读取时，则进行异常处理，因为这说明造成异常的推测指令实际上也是应当执行的（因为它的值被一条非推测指令使用了），这条指令不再处于推测状态。当没有发生异常的指令向有毒寄存器中写入新值时，有毒标记位会被重置，因为这表示这个有毒值没有用，并且该推测指令本不该被执行。图 3-37b 和图 3-37c 中代码的唯一区别是后者的前移 load 指令使

用了推测操作码。如果推测执行的 load 指令没有返回任何异常，或者返回的是对用户不可见的异常，那么硬件就把它当作一条普通的 load 指令。然而，如果它返回的是终止异常，那么就将寄存器 R6 标记为有毒。并且如果分支没有执行，store 指令会捕获到该异常。如果分支执行了，ADDI 指令重新填充 R6 寄存器，并且复位有毒标记位，该异常没有被处理。

有毒寄存器值本身并没有实现精确异常，因为异常在生成异常指令的程序序后的某个时间才被处理。除了违反大多数 ISA 的精确异常模型之外，这种解决方法不会给程序员提供多大帮助，因为异常可能在大量指令执行之后才被发现，并且难以追踪。为了解决这个问题从而实现精确异常，可以在 load 指令的原始执行地方插入一条检测寄存器值的指令，这样就可以在设想的位置触发异常，从而实现精确异常。

如图 3-37d 所示，我们在分支之后插入了一条检测指令，即插入在代码移动之前的 load 指令位置。检测指令只是检测前移的 load 指令的目的寄存器是否中毒。如果是的话，则触发异常并先执行一些修复代码。检查指令有时也称作哨兵（sentinel），哨兵的功能和推测内存歧义消除中的监护指令类似。

3.6 EPIC 微结构

3.4 节和 3.5 节分别介绍了微结构设计中挖掘指令级并行（ILP）的两个极端。一个极端是乱序执行动态调度处理器，它只依靠基于硬件的运行时技术。另一个极端是 VLIW 微结构，所有决策（从取指到完成）都在编译时静态决定。在这两个极端之间的是对静态（编译器）和动态（硬件）支持进行各种权衡折中的设计。

显式并行指令计算（Explicitly Parallel Instruction Computing, EPIC）微结构是在静态和动态机制之间折中的一类结构。EPIC 包含了前面提到的 VLIW 体系结构的大多数概念（实际上前面介绍 VLIW 体系结构的很多机制用的是从 EPIC 微结构中借鉴的术语进行描述的），具体包括：

- 高度依赖编译器技术来分发和调度指令，以满足操作延迟。
- 采用静态预测机制，比如分支预测。
- 循环调度（循环展开和模调度）。
- trace 调度（以及其他类似形式的非循环调度）。
- 解决寄存器重命名的旋转寄存器。
- 谓词执行。
- 针对编译时内存歧义消除的硬件支持。
- 延迟异常。
- 针对推测代码移动的硬件支持。

EPIC 体系结构与 VLIW 体系结构的不同之处在于，EPIC 通过增加硬件支持来协助处理异步事件，例如 cache miss，违反内存别名（歧义消除）和异常等。指令字中的发射槽不依赖于特定的执行单元，并且其类型在不同指令中可能是不同的。另外，指令携带了从编译器传送过来的用于指导和简化硬件的信息，例如指令之间的相关性（用来快速检测寄存器中的数据冲突）和 op 操作的功能单元（用于快速检测潜在的结构冲突，并且避免由于 op 域在指令中不固定所造成的开销）。与 VLIW 体系结构相反，EPIC 机器确实有一些冲突检测 and 解决能力，并且在同一条指令字中分发的 op 操作不需要在整个处理过程中保持同步。一些附加的动态机制，例如动态分支预测等，都可能会影响指令的取指和分发。

这些动态机制通过对常见情况进行调度，并且在硬件层提供保护机制用于检测常见情况下的违例并进行软件恢复，从而达到允许我们采用更激进的编译时调度的目的。这些动态机制的

扩展是可变的，因此软件和硬件之间的边界位置也取决于编译时和运行时机制之间的权衡。例如，对于 cache miss，编译器可能试图在不同的 cache 层次进行 cache miss 的预测，并且基于预测结果进行调度安排，在运行时，如果编译时预测结果错误，那么硬件可以检测到，并且在软件层对调度进行修正。

目前，Intel 安腾系列处理器中采用的 Intel IA-64 指令集就是为 EPIC 架构开发的指令集。

3.7 向量微结构

向量处理器与本章之前提到的所有的微架构都不相同，首先，指令集就与表 3-3 中的指令集有很大的不同。主要的区别在于向量结构的指令操作数是整个向量而不是单独的标量元素，因此实现向量指令集的机器结构也非常不同。

向量机或者向量处理器自 20 世纪 60 年代就已经存在，例如 CDC 的 CDC Star、Cray 向量机以及 IBM3033 的向量单元等。它们那个年代科学和工程计算应用领域最强大的机器，曾被称为超级计算机。今天，向量机依然在超级计算机市场占据主导地位，尽管它们面临着基于数千标量处理器搭建的大规模多处理器的竞争。向量处理器还瞄准了多媒体应用市场，其中大向量的基础数据需要通过流化来处理声音、图像和视频。Alitivec 是 PowerPC 体系结构针对媒体负载处理的一个向量扩展实例。为了充分挖掘向量机的潜力，程序必须先由向量编译器进行向量化。科学、工程、技术以及媒体应用代码中的循环结构为向量化提供了丰富的资源。

同标量机一样，向量机也可能有多种不同的指令集。本节我们将使用一套类似 Cray-1 体系结构的 load/store 向量指令集。load/store 向量结构有一个向量寄存器集，里面的每个向量寄存器都包含大量向量元素。向量操作数必须在操作之前加载到向量寄存器，向量寄存器上的算术和逻辑运算结果被存入另一个向量寄存器。最终，向量寄存器的结果必须写到内存，以便给下一次向量运算腾出寄存器空间。指令有如下类型：

```
ADD.V V1,V2,V3      /* V1=V2+V3, 向量每个位置的对应元素分别相加
L.V V1,R1,R2        /* 从内存地址 (R1) 开始，按照步长 (R2) 加载内容到V1寄存器中
S.V V1,R1,R2        /* 将V1寄存器中的内容存按照起始内存地址 (R1)，步长 (R2) 存储到内存中
```

步长是内存中两个连续向量元素之间的地址差。向量操作受到向量长度寄存器的控制，VL 中保存的是向量元素的数量，它的值必须小于等于所有向量寄存器中元素的数量。

3.7.1 算术/逻辑向量指令

在 load/store 向量体系结构中，所有的算术/逻辑向量操作必须通过向量寄存器完成。从现在开始我们假设所有的向量寄存器最多包括 64 个元素。

图 3-38 给出了流水化的加法功能单元执行 ADD.V 的过程。在 ADD.V 指令译码之后，流水线被配置成按照流模式执行 ADD 指令。两个输入向量寄存器与 ADD 流水线的两个输入相连，ADD 流水线的输出与输出向量寄存器相连。一旦建立完连接，向量寄存器的第一对输入向量元素 (V2[0] 和 V3[0]) 就被送到流水线上，之后其他元素对也依次逐拍送入流水线。经过一定时钟周期 (ADD 流水线深度长度) 后，第一个结果流出流水线，并且存入目的向量寄存器的第一个元素中 (V1[0])。之后，每个时钟周期存储一个向量元素，总共 $N-1$ 个元素， N 是向量的长度。通常，一个向量操作的运行时间可以用如下公式表示：

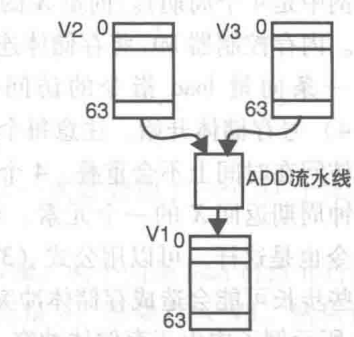


图 3-38 向量 ADD 指令的执行过程

$$T_{ex} = T_{startup} + N \quad (3.1)$$

其中, $T_{startup}$ 是和向量长度无关的开销。

一条向量运算指令 (例如 $ADD.V$) 可以完成最多 64 个标量操作, 它节省了标量处理器中完成相同操作所需的所有处理步骤。此外, 编译器需要确保这 64 个操作数相互独立, 这样就不需要数据前递或者数据冲突检测机制, 也可以确保没有控制冲突。这样, ADD 流水线就可以针对功能进行优化, 也可以增加流水级深度以提高处理速度。深度流水线的唯一缺点在于 $T_{startup}$ 开销, 这个开销必须通过整个向量长度进行分摊。 $T_{startup}$ 越大, 那么短向量操作就越难获得高的效率, 当向量寄存器短到某种程度时, 标量模式执行反而可能会更高效。在某种程度上, $T_{startup}$ 的值决定了向量操作有效的最小向量长度。

3.7.2 内存向量指令

内存向量指令声明了一个基地址和一个步长, VL 寄存器给出了元素长度 (最多 64 个)。对于高效的矩阵操作来说能够以不同步长访问内存这一点很重要。很多科学、媒体和信号处理程序都基于二维或者三维数组, 二维数组必须存到一维线性内存地址中, 一般按行或者按列存储。最起码必须支持对数组行、列和对角线的高效访问。

图 3-39 给出了对一个 $N \times N$ 矩阵的主要访问类型。假设数组在内存中按行存储, 那么行以步长 1 访问, 列以步长 N 访问, 正向对角线以步长 $N+1$ 访问, 反向对角线以步长 $N-1$ 访问。

由于计算在向量中的结构化, 每个内存访问指令实际上是对大量 (一次最多 64 个) 内存地址访问。更重要的是, 向量所有元素的位置在指令译码阶段就是已知的。这与标量机器不同, 标量机器的硬件每次计算一个内存地址。

因此, 内存向量指令中的所有内存访问在指令译码之后就都可以立即调度。向量机不需要 cache, 因为 cache 的行为不可预测。内存组织针对按照固定步长读取这种模式进行优化。内存通常组织成交叉的、独立的多个存储体 (bank), 以便连续的向量元素分布在不同的存储体中, 从而可以并行访问。虽然内存访问时间较长, 但是向量模式下, 数以百计的存储体可以同时激活并向处理器高速传输所有的向量元素。

图 3-40a 显示了一个用来进行高效向量访问的交叉存储组织结构。连续的地址分布在连续的存储体中以便可以并行访问。存储体的数量等于访问存储体所需的时间 (以时钟周期为单位, 本例中是 4 个周期)。向量 X 的元素交叉存入存储体中, 因此 $X[i]$ 被存入 $(i \bmod 4)$ 号存储体。内存控制器 MC 将存储体连接到向量寄存器, 并控制对存储体访问的顺序。图 3-40b 展示了一条向量 load 指令的访问时间。在连续的多个时钟周期中, 对 $X[i]$ 的访问从 $(i \bmod 4)$ 号存储体开始。注意每个存储体只在 4 个时钟周期中是繁忙的, 这样对相同存储体的连续访问在时间上不会重叠。4 个时钟周期之后, $X[0]$ 的值由 0 号存储体返回。紧随其后, 每个时钟周期返回 X 的一个元素。对向量 load 指令访问的调度与逻辑算术指令的调度相似, store 指令也是这样。可以用公式 (3.1) 计算执行一条向量 load 或者 store 指令所花费的时间。通常有些步长可能会造成存储体冲突, 从而减慢内存取数的速率。例如, 如果步长为 2 或者 4, 图 3-40 所示例子中由于存储体冲突, 访问速率将变成每个元素需要 2 个或者 4 个时钟周期。存储体越多, 则冲突的可能性越小。而存储体的最小数量取决于单个存储体的访问周期数。当然, 为了并行执行不同向量中的多个不冲突的 load 和 store 操作, 增大存储体数量也是必要的。

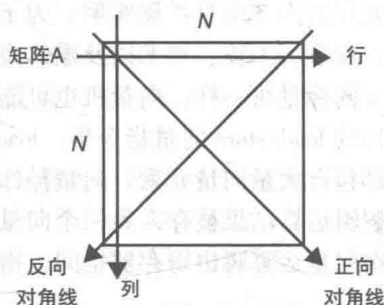


图 3-39 $N \times N$ 矩阵的访问类型

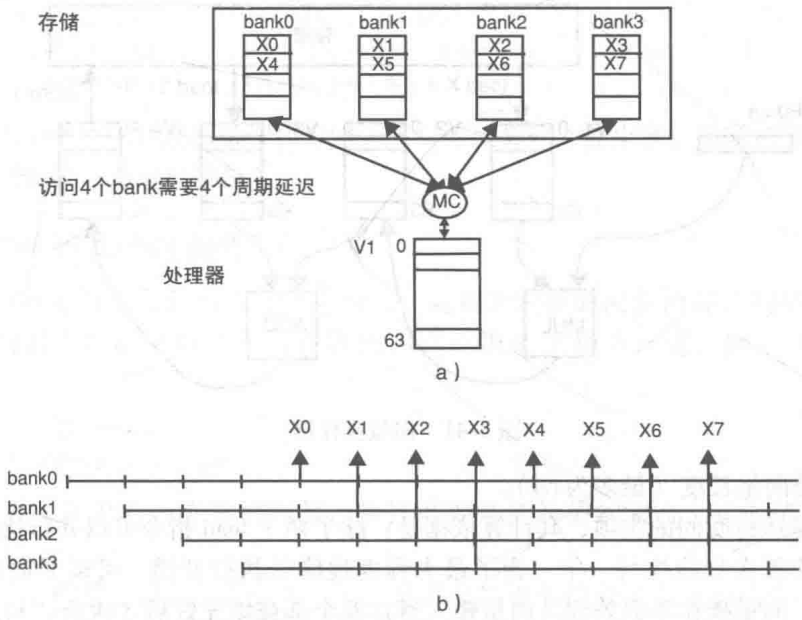


图 3-40 向量访问的存储组织方式

3.7.3 向量分段开采和链接

在 load/store 体系结构中，向量操作数必须首先放到寄存器中。由于向量寄存器有长度限制（这里是 64 个元素），因此较长的向量操作数必须被分割为适合寄存器长度的连续分段（strip），这个过程被称为向量分段开采。

考虑如下的长向量操作：

$$Y = a X + Y$$

其中，X 和 Y 是内存中的浮点长向量，同时 a 是内存中的一个浮点标量。在 load/store 体系结构中，支持长向量操作的代码是一个在每次迭代中处理 64 个向量元素的循环：

```

L.D      F0,0(R1)          /* 将标量元素读入F0
LOOP:    L.V      V1,0(R2),R6    /* 读取X的64个元素到 V1，步长为1 ((R6)=1)
        MUL.V     V2,V1,F0
        L.V      V3,0(R3),R6    /* 读取Y的64个元素到V3，步长为1
        ADD.V     V4,V2,V3
        S.V      V4,0(R3),R6    /* 将Y的64个元素存入 V4，步长为1
        ADDI     R2,R2,#64
        ADDI     R3,R3,#64
        ADDI     R4,R4,#1
        BNE      R4,R5,LOOP

```

这个循环看起来像是标量机里的循环，但是循环每迭代一次就处理了一个 64 个元素的向量段。现在用图 3-41 来说明一个向量段的执行过程。

该图给出了循环中每次迭代时的向量操作，包括 4 个步骤。两个向量 load 可以并行执行读取数据到 V1 和 V3 中，然后使用 MUL.V 指令将 V1 乘以标量 a，结果存入 V2。然后再使用 ADD.V 指令将 V2 和 V3 相加，结果存入 V4。最后将 V4 存入内存。假设访问内存没有冲突，64 个元素的向量段总共运行时间为：

$$T_{ite} = T_{startup}(L) + T_{startup}(MUL) + T_{startup}(ADD) + T_{startup}(S) + 4 \times (VL) \quad (3.2)$$

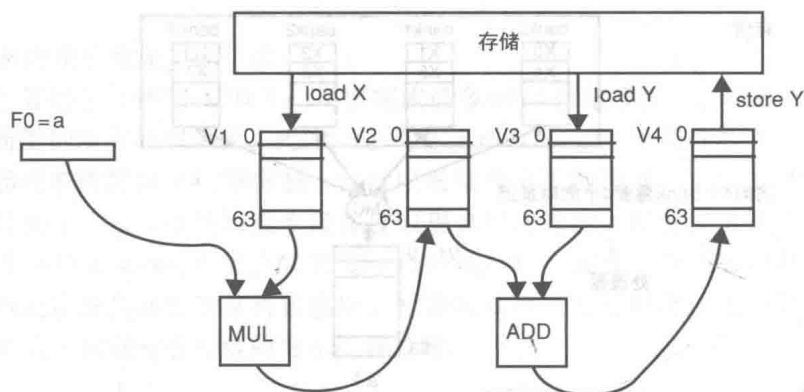


图 3-41 向量操作链

其中, (VL) 是向量长度 (最多为 64)。

最后一项是运行时间的主项, 其计算依据是: 除了两个 load 指令可以并行执行外, strip 中的其他向量操作每次只能执行一个。为了最大程度地降低执行开销, 可以对向量操作进行链接。链接之后, 向量操作不会等到其向量输入寄存器全部被填充好后才执行, 只要它的输入寄存器的第一个元素可用, 向量操作就会立即开始执行, 这样两个操作就可以在一个长的组合向量操作中链接起来。每个输入向量寄存器有一个写端口和一个读端口。读端口上的寄存器地址会对写端口上的寄存器地址进行跟踪, 以便数据一写入寄存器堆就可以立即使用。理想情况下, 如果一切顺利, 多个向量操作会转化为一个包含多级流水的长向量操作。

例如, 图 3-41 中, 只要 load X 的第一个元素存入 V1[0], MUL V 就开始执行。之后, 当 MUL V 的第一个结果存入 V2[0] 时, ADD V 立即开始执行。注意 ADD V 的第二个输入 (load Y 的结果) 还在计算中, 但是这不重要, 因为 V3 的写端口和读端口不需要访问连续的寄存器。重要的是读端口在写端口之后。最后, 当 ADD V 的第一个结果可用并存入 V4[0] 时, store Y 也开始执行。如果一切顺利 (没有内存冲突), 采用链接技术的分段执行时间可以如下计算:

$$T_{ite} = T_{startup}(L) + T_{startup}(MUL) + T_{startup}(ADD) + T_{startup}(S) + (VL) \quad (3.3)$$

对于 (VL) = 64 来说, 节省的时钟周期总数为 192, 这是一个巨大的性能提升。

3.7.4 条件语句

包含条件语句的循环无法通过前面介绍的机制进行向量化。带有条件语句的循环可以通过谓词替换条件分支来向量化, 就像 VLIW 和 EPIC 中采样的方法。通过执行一个类似标量机环境中的 MIPS SLT 指令那样的逻辑向量操作, 将谓词存储到向量掩码 (Vector Mask, VM) 寄存器中。然后, 向量操作可以在向量掩码的控制下执行。考虑如下的循环:

```
for(i=0; i<64; i++)
    if (A[i]==0) then A[i]=B[i];
    else A[i]=A[i]+x;
```

这段代码的汇编版本 (向量化的) 首先对循环进行分段开采。通过将 A 中的元素和 0 进行比较得到向量掩码。对于每个段, 代码首先计算 then 子句内的语句 (这里是一条简单的 load 指令) 和 else 子句内的语句, 然后 then 和 else 子句的结果在向量掩码的控制下进行合并。

```
L.V      V1,0(R1),R6    /*将A按步长1读入到V1
SETMZ    V1              /*如果A[i]=0, 将VM[i] 设置成1
```

```
L.V      V2,0(R2),R6    /*将B按步长1读取到V2
ADD.V    V2,V1,F0,VM    /*根据VM中的谓词,将A和x相加,结果存入V2
S.V      V2,0(R1),R6    /*将V2按步长1存入A
```

向量操作按照 VM 的值进行条件执行时，所有的值都要计算出来，但是结果是否要存入目的向量寄存器则取决于向量掩码位。

3.7.5 scatter 和 gather 操作

当访问模式按照固定步长有规律地进行时，向量处理器的向量内存访问是最高效的。然而，有时候，我们需要从分散的不同位置的内存中取值来构造向量，这个过程叫作 gather 操作：

```
for(i=0;i<N;i++)
    A[i]=B[INDEX[i]]
```

INDEX 是向量 B 中的一个索引向量，并且可以是任何值的集合，这些值在编译时是未知的。

与之相反，有时候，我们也需要将一个向量寄存器中的值分散存储到内存的不同位置中，这个过程叫作 scatter 操作。

```
for(i=0;i<N;i++)
    B[INDEX[i]]=A[i]
```

某些向量处理器没有实现 gather 和 scatter 操作的向量指令，这种情况下，通过标量模式执行。图 3-42 说明了 gather 和 scatter 向量指令的功能。在 gather 指令中（左侧），向量寄存器 V1 被来自内存中不同地址的值填充。这一过程可以通过以另一个向量寄存器的元素作为索引的特殊 load 指令来实现。在 scatter 指令中（右侧），向量寄存器 V2 的内容被存入内存中不同的地址。这一过程也可以通过一条特殊的 store 指令实现，该 store 使用存储在另一个向量寄存器中的索引值作为地址。

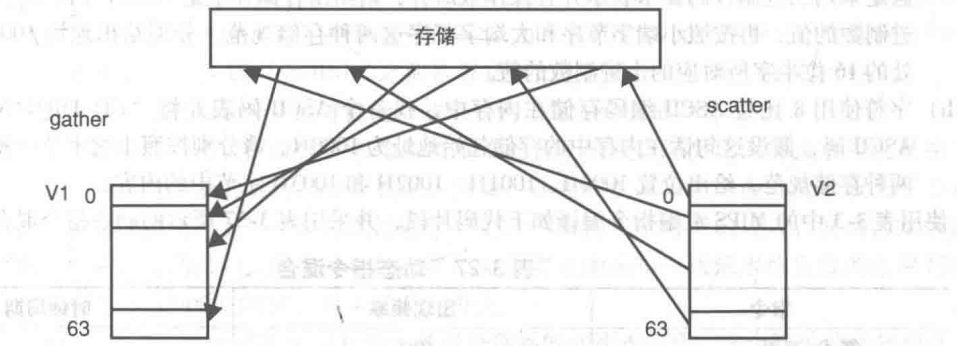


图 3-42 向量 gather 和 scatter 操作实现

gather 和 scatter 操作的一个重要应用是稀疏矩阵计算。稀疏矩阵计算中存在大量零元素，如果像稠密矩阵一样按行存取，并且处理矩阵中的所有零元素的话，那将极大浪费内存和处理器带宽。节省内存和避免无用操作的方法之一是将矩阵表达形式从稠密变换为稀疏。在稀疏矩阵表达形式中，矩阵 A 以两个向量进行存储：一个存储矩阵 A 中所有非零元素的向量 A_VALUE，一个存储矩阵 A 中这些非零元素索引值的向量 A_INDEX，即 A_INDEX[i] 是矩阵 A 中非零元素 A_VALUE[i] 的索引。gather 操作将矩阵表达形式从稠密转换为稀疏，反之，scatter 操作将矩阵表达形式从稀疏转换为稠密。

习题

3.1 对比以下四种指令集架构 (ISA) 在代码紧凑性和访存流量两方面的效率 (参见 3.2.3 节):

- 基于累加器 (accumulator-based)。
- 基于栈 (stack-based)。
- 基于内存到内存 (memory-to-memory, 所有的操作数都位于内存)。
- 基于寄存器 (register-based) (纯 load/store)。

假定指令和数据大小如下:

- 所有的数据操作数长度都是 4 字节。
- 所有的内存地址都是 16 位。
- 所有操作码长度都是 1 字节。
- 所有的内存地址字段长度都是 2 字节。
- 在 load/store 类型机器中, 所有寄存器字段长度都是 4 位 (支持 16 个 32 位寄存器)。

此外, 所有的内存地址都是 32 位, 所有的指令和数据取数都可以在一次访存中完成。考虑如下的 HLL 代码:

```
A = B + A
C = A - C + D
```

使用上述四种不同指令集架构 (基于累加器、栈、内存到内存, 以及 load/store 型) 编译此代码。其中在内存到内存架构中, 不需要使用额外的内存位置。对于编译生成的每段代码, 请确定以下指标: (1) 代码大小; (2) 数据访存流量 (包含寻址); (3) 指令流量 (包含地址)。然后基于这三个指标对上述四种指令集架构进行对比。

3.2 (a) 从地址 1000H 到地址 1003H 中的字节填充为如下数字:

```
1000H: 23H
1001H: F7H
1002H: 32H
1003H: AB
```

假定采用二进制补码算术表示并且操作数对齐, 请给出存储在地址 1000H 中的 32 位字对应的十进制数的值, 再按照小端字节序和大端字节序这两种存储规范, 分别给出地址 1000H 和 1002H 处的 16 位半字所对应的十进制数的值。

(b) 字符使用 8 比特 ASCII 编码存储在内存中。找一个 ASCII 码表并将 “GO TROJANS!” 翻译成 ASCII 码, 假设这句话在内存中的存储起始地址为 1000H。请分别按照小端字节序和大端字节序两种存储规范, 给出位置 1000H、1001H、1002H 和 1003H 字节中的内容。

3.3 使用表 3-3 中的 MIPS 汇编指令编译如下代码片段, 并采用表 3-27 所示的动态指令混合方式。

表 3-27 动态指令混合

指令	出现频率	时钟周期
算术/逻辑	40%	1
Load	25%	2
Store	10%	1
分支 (不跳转)	8%	1
分支 (跳转)	12%	3
其他	5%	1

```
S = 0;
for (i=0; i<100; i++)
    S = S + A[i];
```

数组 A[i] 由 100 个 4 字节的整型数组成，存储在连续递增的内存地址中，A[0] 存储在内存地址 1000 中。除了以下限制，你可以随意分配寄存器。

- (a) 第一，假设在整个循环中，没有变量（包括 i）会被编译器分配到寄存器，即所有的变量在使用时都从内存中读入（装入时）或者写入到内存（溢出时）。请给出代码，并根据表 3-27 中每类指令的执行时间来估计整体执行时间。
- (b) 第二，假设在整个循环中，S 和 i 都被分配在寄存器中。一开始它们就载入到寄存器中并完成初始化。然后在循环结束时，寄存器溢出写入内存中。其他的内存值在需要时也先读入寄存器。请给出代码，并根据表 3-27 中每类指令的执行时间来估计整体执行时间。

3.4 根据 3.3.1 节的经典流水线设计，考虑如下的程序，该程序对一块内存区域进行查找，并计算和关键字匹配的内存字的次数：

```
SEARCH:    LW R5,0(R3)      /I1    载入项
           SUB R6,R5,R2     /I2    对比关键字
           BNEZ R6,NOMATCH  /I3    检查是否匹配
           ADDI R1,R1,#1    /I4    计算匹配次数
NOMATCH:   ADDI R3,R3,#4    /I5    下一项
           BNE R4,R3,SEARCH /I6    继续直到所有项检查完毕
```

假定分支总是预测为不跳转，如果实际跳转的话，需要到 EX 流水级才能确定。下面的所有情况都包括支持分支的硬件结构。考虑如下几种解决数据冲突的可能流水线互锁机制，并针对每个循环迭代（除了最后一次迭代外）回答以下问题：

- (a) 首先，假设流水线没有前递单元和冲突检测单元，甚至在寄存器堆内部也不支持值的前递。请重写上述代码，在需要的地方插入 NOOP 指令以使代码能够正确执行。
- (b) 接着，假设仍然不支持前递，但冲突检测单元会在 ID 阶段暂停指令以避免冲突。在下列情况下循环执行一次迭代需要耗费多少时钟周期：（1）匹配的情况下；（2）没有匹配的情况下。
- (c) 接着，假设支持寄存器前递，并且冲突检测单元会在 ID 阶段暂停指令以避免冲突。在下列情况下循环执行一次迭代需要耗费多少时钟周期：（1）匹配的情况下；（2）没有匹配的情况下。
- (d) 接着，假设支持完全前递，并且冲突检测单元会在 ID 阶段暂停指令以避免冲突。在下列情况下循环执行一次迭代需要耗费多少时钟周期：（1）匹配的情况下；（2）没有匹配的情况下。
- (e) 请找出代码中的基础块（使用指令编号表示）。如果采用局部优化是否可以缩短执行周期？为什么？如果编译器将 I5 移到 BNEZ 之前是否安全？这样做是否有效？效果如何？
- (f) 编译器可否通过循环展开来改善性能？效果如何？延迟分支是否有效？

3.5 现在探讨流水线设计。流水线设计需要均衡流水线的各个阶段，完美均衡的流水线就是指完成流水线的每一个阶段所耗费的时间都完全相同。然而，在实际中，完美均衡的流水线几乎不可能存在。因此，良好的流水线实现，首先就是创建一个简单而且几乎均衡的设计，使得流水线的每个阶段都耗费大致相同的时间。流水线设计的第二个重要方面是每一级流水线生成的结果都会被下一级使用。在回答本问题的时候，请牢记以上两点。

图 3-43 显示了一个只能执行 ALU 指令的微处理器的非流水实现。这个简单的微处理器需要执行一系列任务：首先，它通过递增 PC 的值来计算下一条要取指令的地址。其次，它使用 PC 访问 l-cache。然后，再对指令进行译码。指令译码器本身又被分成多个更小的任务，要译码出指令类型，一旦操作码译码完成，它需要译码出执行指令所需的功能单元，同时，它也会译码出指令所使用的源寄存器或立即数和需要写入的目标寄存器。一旦译码过程完成，就将访问寄存器堆（立即数可从指令自身访问），以获得源数据。接着，激活对应的 ALU 单元（由功能译码器指定）用于计算结果，结果再被写回到目标寄存器中。

请注意这个微处理器的译码机制与之前讨论的 MIPS 译码器不同。在 MIPS 译码器中，译码是整体作为一个单一的操作，即译码器在进行源寄存器译码之前不需要知道操作码。然而，在这个简单微处理器中，译码器需要在译码源操作数和目标操作数之前就知道操作码。此外，译码源操

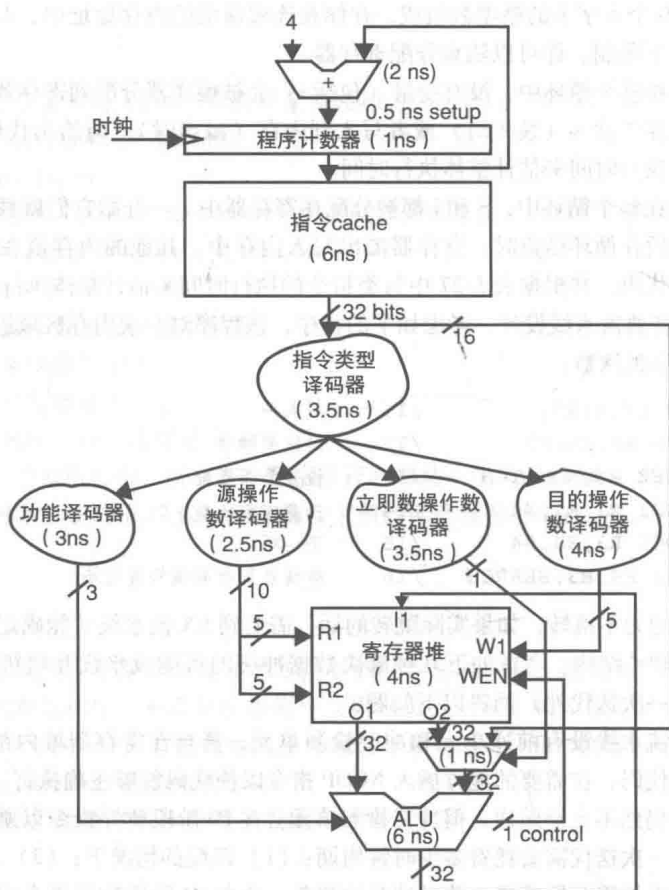


图 3-43 寄存器-寄存器指令的非流水机器

作数和目标操作数的任务实际上也被细分成多个更小的任务。

请注意每一个模块的延迟在图中已经标注出来。例如，访问 I-cache 需要花费 6ns，访问寄存器堆需要 4ns 等。

(a) 针对上面所介绍的简单处理器，实现一个五级流水线 (IF, ID1, ID2, EX, WB)，尽量使所有的流水级之间保持均衡，不考虑任何的数据冲突。图 3-43 中的每一个子块都是一个不可再分的基本单元，每个流水线寄存器需要 0.5 ns 的建立时间。在流水实现中必须维持原有的功能，换句话说，程序员编写代码时，和这台机器是否支持流水线无关。请给出流水线实现的示意图。

(b) 非流水和流水实现的指令周期的延迟 (以 ns 为单位) 分别是多少？假定在流水实现中，每个流水线寄存器需要 0.5 ns 的建立时间。

(c) 非流水和流水实现的机器周期时间 (以 ns 为单位) 分别是多少？

(d) 流水实现与原来的非流水实现相比，可获得的加速比是多少？

(e) 哪种微架构技术可以进一步减少流水设计中的机器周期时间？找出瓶颈所在，说明机器周期时间是如何减小的。

3.6 重复问题 3.4，但假设 CPU 是超流水结构的，所以取指令需要两个周期 (IF1 和 IF2)，数据 load 和 store 也需要两个周期 (ME1 和 ME2)，因此，流水线总共有 7 级。假定分支指令在 EX 阶段中处理并且总是硬件预测为不跳转。

3.7 在经典的五级流水线中，假定将分支结果总是预测为跳转而不是不跳转。分支指令在 ID 阶段进行译码并计算其目标地址，在 ID 阶段结束时，条件分支总是跳转，同时刷掉 IF 中的内容。然后，在 EX 阶段中，对分支条件进行确认。如果确认分支是跳转的，则继续执行。否则，如果确认分支是

不跳转的, 则刷掉 IF 和 ID 流水级, 并且重新到 $\text{branch_PC} + 4$ 处取指执行。

(a) 跳转的分支占分支指令的比例 f 应该达到多少时, 才能使始终跳转的分支预测设计相比始终不跳转的分支预测设计更好?

(b) 在每个条件分支指令中增加一个提示位。编译器通过设置提示位引导硬件预测为跳转, 通过重置提示位引导硬件预测为不跳转。提示位在译码阶段进行识别, 因此, 实现这两个硬连线方案 (始终跳转和始终不跳转) 在分支指令执行过程中不会引入额外的周期损耗。请问, 编译器应该获得多少的预测成功率, 才能使这种方法的性能总是比分支始终预测为跳转的硬件方案要好? 如果要比分支始终预测为不跳转的硬件方案更好, 编译器的预测成功率又应该是多少? 假定编译器对跳转分支和不跳转分支的预测成功率相等, 请用以下变量描述你的解决方案:

- f 是跳转分支所占的比例。
- X 是编译器预测算法的成功率 (即被编译器准确预测分支结果的分支指令比例); X 应该是 f 在这两种情况下的函数。

(c) 以支持完美分支处理 (最优机制, 分支不会浪费任何的时钟周期) 的实现作为比较基准, 请在此之上比较以下几种情况对应的单条指令能耗 (Energy Per Instruction, EPI):

- 总是预测不跳转。
- 总是预测跳转。
- 带有提示位的编译分支预测, 且错误预测率为 5%。

为了简化问题, 假定每一级流水线在每一个时钟周期的能耗相同 (无论是什么指令, 即使 NOOP 指令也是这样), 刷流水线的能耗忽略不计。假设指令中分支指令所占的比例为 b 。

3.8 本题中, 我们评估在支持指令乱序执行完成的不同静态流水线中, 支持冲突检测所需的硬件结构。

在图 3-8 所示的五级流水线基础上增加浮点扩展。

每个流水线寄存器 (浮点或者整型) 都含有自己的目标寄存器号, ME/WB 执行两条指令, 一条来自整型流水线, 一条来自浮点流水线。

考虑如下的指令类型:

- 整型算术/逻辑/存储指令 (输入: 两个整型寄存器) 和所有 load 指令 (输入: 一个整型寄存器)。
- 浮点算术指令 (输入: 两个浮点寄存器)。
- 浮点 store 指令 (输入: 一个整型和一个浮点寄存器)。

所有值都尽可能早地进行前递, 整型和浮点这两个寄存器堆都支持内部前递。所有数据冲突都在含有冲突检测单元 (HDU) 的 ID 阶段解决。ID 根据需从整型或浮点寄存器堆中读取寄存器, 由操作码来确定操作数取出的寄存器堆。

(a) 为了解决在寄存器中的 RAW 数据冲突 (整型或浮点数), 硬件会检查 (互锁) ID 中的当前指令和流水线中可能会阻塞 ID 阶段指令的其他指令。首先请列出所有需要在 ID 阶段进行检查的流水线寄存器。由于 ME/WB 可能有两个目的寄存器, 将它们表示为 ME/WB (int) 或 ME/WB (fp)。不需要列出各个流水线, 列出流水线寄存器, 并确保该组检查规模是最小的。

(b) 为了解决寄存器上的 WAW 冲突, 将 ID 阶段的目的寄存器与各个流水级中指令的目标寄存器进行对比检查。请列出所有需要检查的流水线寄存器, 并确保这是需要检查的最小组合。重要提示: 请记住在 ID 阶段有相关机制可以避免两个寄存器堆的写寄存器端口的结构冲突。

解决 RAW 和 WAW 冲突的方案请按照如下格式表述:

- 如果是 ID 阶段的整型算术/逻辑/store/load 指令, 检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP load 指令, 检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP 算术指令, 检查 <流水线寄存器>。
- 如果是 ID 阶段的 FPstore 指令, 检查 <流水线寄存器>。

3.9 使用如图 3-10 所示的超流水架构重复问题 3.8。假设所有指令都支持前递 (包含 FP store)。需要

注意的是浮点和整型的值可以从 ME1/ME2 和 ME2/WB 中进行前递。

解决 RAW 和 WAW 冲突的方案请按照如下格式表述：

- 如果是 ID 阶段的整型算术/逻辑/store/load 指令，检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP load 指令，检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP 算术指令，检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP store 指令，检查 <流水线寄存器>。

3.10 如图 3-9 所示的流水线可以消除寄存器上的 WAW 数据冲突，并且可以在 WB 流水级处理异常，因为指令和经典五级流水线一样按照程序顺序完成。与之前的假设一样，值可以前递到执行单元的输入端口。

(a) 为了支持所有指令的完全前递，请列出从流水线寄存器到 EX 或者 FP1 所需的所有前递通路，按照“源→目的（如，FP2/FP1→FP1）”这样的格式列出。

(b) 基于上述前递通路，如果要解决 RAW 冲突，请给出在 ID 阶段的冲突检测单元（HDU）需要做的所有检测。请按照如下格式给出解决寄存器 RAW 冲突的方案：

- 如果是 ID 阶段的整型算术/逻辑/store/load 指令，检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP 算术指令，检查 <流水线寄存器>。
- 如果是 ID 阶段的 FP store 指令，检查 <流水线寄存器>。

(c) 这种架构在异常处理上还存在一个小问题，那就是 store 执行较早并且在写回流水级 retire 之前就改写了内存，请问如果这样做，那么会存在什么问题？请给出一个解决该问题的方案。（不建议类似“存储内存值，然后在异常中进行恢复”这样的解决方案。）

3.11 考虑图 3-44 所示的超标量体系结构，该结构可以同时取两条连续指令，PC 递增 8。为了简化流水线互锁，把译码拆分为 ID1 和 ID2 两个阶段。通过一个支持两种设置（直接连接和交叉连接）的开关将 ID1 和 ID2 分开。上层 ID2 只能处理整型/分支指令或 FP load/store 指令。下层 ID2 只能处理 FP 算术指令。

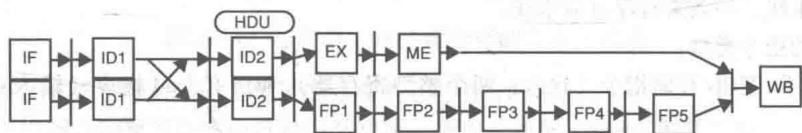


图 3-44 两路超标量 CPU

假定 I1 是 ID1 中的上层指令，I2 为 ID1 中的下层指令。为了保证程序序，I1 必须在 I2 之前、或者是同时进入 ID2，以便保持处理顺序。以下是在 ID1 完成的操作。

- 如果 I1 为一个整型/分支指令或 FP load/store 指令或者是 NOOP，并且 I2 不依赖于 I1（预期的情况下），那么将开关切换到直接连接。
- 如果 I1 是一个 FP load 指令并且 I2 需要使用由 load 返回的值，在 ID1 中阻塞住 I2，并设置开关为直接连接（下层 ID2 送入 NOOP 指令）以便将 I1 发送至 ID2。
- 如果 I1 是 FP 算术指令，阻塞 I2 并将 I1 发送至 ID2，设置开关为交叉连接（上层 ID2 送入 NOOP 操作）。
- 如果 I1 和 I2 都是整型/分支指令或 FP load/store 指令，在 ID1 上阻塞 I2 并移动 I1 至 ID2，设置开关为直接连接（下层 ID2 送入 NOOP 操作）。
- 如果 I2 是整型/分支指令或 FP load/store 指令，并且 I1 是 NOOP，将 I1、I2 发送到 ID2，设置开关为交叉连接。

因此，如果取到的两个指令存在依赖或者类型不匹配，它们就会在 ID1 串行化操作。由于流水线冲突，ID2 中的指令会被阻塞，就好像是一个单发射处理器。如果它们与流水线中之前的其他指令都没有数据冲突，则可以继续往前推进。我们实现如图 3-8 所示的前递通路，当指令在 ID2 上阻塞时，则在 IF 和 ID1 上的指令也同样会阻塞。

- (a) 简要描述 ID2 对应的 HDU 的功能。
- (b) 假定分支总是硬件预测为不跳转，请说明分支指令的处理过程（考虑分支在 ID1 上层或下层的两种情况）。
- (c) 考虑下面的代码：

```
LOOP      L.D F2,0(R1)
          ADD.D F4,F2,F4
          L.D F6,-8(R1)
          ADD.D F8,F6,F4
          S.D F8,0(R1)
          SUBI R1,R1,16
          BNEZ R1, LOOP
```

比较该循环的一次迭代（不包括最后一次迭代）在本题对应机器以及在习题 3.8 和 3.9 所描述的机器上的执行时间。假定习题 3.9 的机器可以在 EX1 阶段确定分支指令结果。

3.12 借助深度流水线和较高的时钟频率，超流水线技术看起来是提高性能的一种可扩展解决方案。不过，这项技术仍然有以下几个问题：

- 当流水级数增加时，功能逻辑的延迟将按比例下降，但流水线寄存器的延迟不会改变。
- 数据依赖所引发的开销（流水线空泡）将会增加。
- 当分支预测错误时，刷掉的流水级数会增加。
- 在深度流水线下，如果内存性能没有提高，那么 cache 失效的代价将会增大。

我们按照如下方式进行建模，令 T 是单周期 CPU 的时钟周期， K 为流水级数。使用 K 级流水线的 CPU 的时钟周期按如下方式建模：

$$T_K = \frac{T}{K} + t_1 = \frac{1}{f_K}$$

其中 t_1 是锁存每个阶段输出所需的时间（建立时间）。数据冲突导致的每条指令代价（以时钟周期为单位）模型为：

$$\Delta_{data} = \alpha_d \frac{K}{5}$$

同样，由于错误预测分支（控制冲突）导致的每条指令代价建模为：

$$\Delta_{branches} = 2\alpha_b \frac{K}{5}$$

最后，因为 cache 失效时需要更多的时钟周期，所以 cache 失效的代价也会影响深度流水线的加速比，通过以下方式建模：

$$\Delta_{memory} = \alpha_m \frac{K}{5}$$

α_d 是由于五级流水线的数据冲突，每条指令在 ID 阶段的平均阻塞数； α_b 是预测错误的分支指令所占比例（假设在五级流水线中预测错误的代价是两个时钟周期）； α_m 是在五级流水线中，由于 cache 失效而导致的每条指令平均浪费的时钟周期数。

- (a) 解释这些模型的基本原理。（提示：上述模型大致基于五级流水线中的代价）
- (b) 给出一个以流水级数 K 为变量的计算预期指令吞吐量的公式。
- (c) 是否存在一个可获得最佳吞吐量的最优流水线深度？如果有，最优流水线深度是多少（表示为 t_1 、 α_d 、 α_b 和 α_m 的函数）？
- (d) 在一个实际案例中， $\alpha_d = 0.2$ （由于寄存器 RAW 冲突，每 5 条指令中会有 1 条有一个周期的延迟）， $\alpha_b = 0.06$ （五分之一的指令是分支指令，静态分支预测成功率为 70%）， $\alpha_m = 0.5$ （每条指令平均对应 0.05 次的 cache 失效，cache 失效的开销为 10 个周期）。再假设单周期 CPU 的指令延迟时间 T 为 10ns，流水线寄存器的开销是 100ps。请问最优的流水线深度是多少？在上述最优流水线深度下的吞吐量是多少？在相同的假设下，它与五级流水线相比如何？

(d) 根据单个迭代的数据流图中的关键路径延迟，计算最小可能的执行时间。数据流图中的每个节点是循环迭代中的一条指令，节点之间通过有向边连接，每条有向边对应两条 RAW 相关的指令（父指令和子指令）。有向边上标记了父指令的执行时间（以周期为单位）。这里只考虑数据依赖（假设硬件资源无限，cache 100% 命中，分支预测总是正确）。

请画出对应循环一次迭代的数据流图，在数据流图上确定关键路径，并计算数据流图计算出的最好的可能执行时间。比较在上面三种不同的结构下循环第一次迭代所需的执行时间。可以通过比较相邻两个迭代内的第一个 load 操作发射的时钟周期之差来计算循环的执行时间。

3. 14 本题比较复杂，我们将探讨推测执行中的一些在之前没有讲过的技术，包括多指令分发和 ROB 中的结构冲突。

为了简化起见，我们使用与习题 3. 13（b）相同的架构，即支持推测的 Tomasulo 算法，ROB 的作用是保存推测值，并跟踪指令的线程序。

每个时钟周期分发两条指令。ROB 尺寸为 8 项，当 ROB 满时停止分发。ROB 中必须有两项空闲时，才会执行两条指令的分发，从而确保指令的分发总是成对进行。

在分发这一列中记录了指令分发所在周期（括号中的数）结尾留在 ROB 中的项数。每次分发一条新指令时，在下一个周期就会占用一个新的 ROB 项。当指令进入 retire 阶段时，同一周期内对应的 ROB 项就会被释放，并且立即提供给其他新的指令。

为了观察 ROB 冲突的影响，跟踪两个循环迭代。请完成表 3-31（表的前两行已经填好）。

表 3-31 支持推测的 Tomasulo 算法（两路超标量）

	分发	发射	开始执行	完成执行	cache	CDB	retire	备注
I1 L D F0, 0(R1)	1 (7)	2	(3)	3	(4)	(5)	6	
I2 L D F2, 0(R2)	1 (6)	3	(4)	4	(5)	(6)	7	

类似前面的习题，通过比较第二个和第三个迭代中各自第一个 load 发射的时钟周期之差，可以估算出一次循环迭代的执行时间。请问两路分发是否提高了性能？瓶颈在哪里？

3. 15 在本题中，我们探讨使用简单的内存 move 操作来消除内存歧义的效果：

```
for(i=0;i<100;i++)
    A[i] = B[i];
```

在上述代码中，向量 A 和 B 在内存的不同区域，两者没有共同的元素。对应的汇编代码如下：

```
LOOP      L.D F2,0(R1)
          ADDI R1,R1,#8
          ADDI R2,R2,#8
          S.D F2,-8(R2)
          BNEQ R1,R3,LOOP
```

采用和习题 3. 14 相同的架构（支持推测和两路分发的 Tomasulo 算法）。根据如下两种不同情况分别填充表 3-32：（1）保守模式（直到之前所有 store 的地址都已知时才会将 load 操作发射到 cache）；（2）推测模式（当之前 store 的地址未知时，load 操作也会发到 cache）。请记住，store 操作必须等到达 ROB 顶部时才能发射到 cache。

表 3-32 支持推测的 Tomasulo 算法（两路超标量）

	分发	发射	开始执行	完成执行	cache	CDB	retire	备注
I1 L D F2, 0(R1)	1 (7)	2	(3)	3	(4)	(5)	6	
I2 ADDI R1, R1, #8	1 (6)	2	(3)	3	—	(4)	7	
I19 L D F2, 0(R1)								

3.16 考虑循环中如下代码片段:

```

if (x is odd) then          <- (分支b1)
    increment a             <- (b1不跳转)
if (x is a multiple of 5) then <- (分支b2)
    increment b             <- (b2不跳转)

```

假设用上述循环的 9 次迭代分别处理如下的 x 值序列: 8, 9, 10, 11, 7, 20, 29, 30, 31。

- (a) 假设用 1 位的状态机 (见图 3-45a) 来预测循环中的两个分支执行。给出循环每次迭代中 $b1$ 和 $b2$ 这两条分支指令的预测结果和实际分支方向。假设预测器的初始状态为 0, 即不跳转 (not taken, NT)。 $b1$ 和 $b2$ 的预测准确度分别是多少? 两个分支的整体预测准确度又是多少?

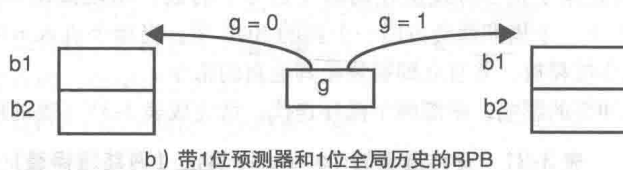
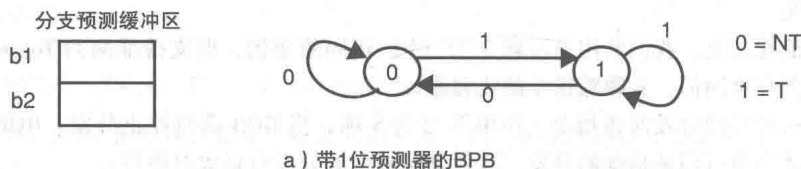


图 3-45 分支预测缓冲区 (BPB)

- (b) 假设现在使用一个两级的分支预测方案。除了 1 位预测器之外, 还使用一个 1 位全局历史寄存器 (g), g 存储上次执行的分支方向 (可能与当前预测的分支是同一条指令, 也可能不是), 并用于索引两个独立的 1 位预测器表, 如图 3-45b 所示。

根据 g 的值, 选取这两个预测器表中的一个用于通常的 1 位预测。此后, 按照相同的方式, 填写循环 9 次迭代中所有 $b1$ 和 $b2$ 的预测分支方向和实际分支方向。假设 g 的初始值为 0, 即 NT。每次预测时, 根据 g 的当前值, 两个预测表中只有一个会被访问并更新。

对于循环的每次迭代, 请给出 g 的值, 以及 $b1$ 和 $b2$ 分支指令的预测分支方向和实际分支方向。假定预测表的初始状态全部为零, 那么 $b1$ 和 $b2$ 的预测准确度分别是多少? 所有分支的整体预测准确度又是多少?

- (c) 当 $g=0$ 时, $b2$ 分支的预测成功率是多少? 请解释原因。

3.17 考虑下面的循环:

```

                ADDI R1,R0,#1000
                ADD R2,R0,R0
LOOP:          BNEZ R2,LAB1
                ADDI R2,R0,#1
                BEQZ R0,LAB2
LAB1:          ADD R2,R0,R0
LAB2:          SUBI R1,R1,#1
                BNEZ R1,LOOP

```

对于每种类型的分支预测机制 (GAg, GAp, PAg, PAp), 使用大小为 1 的分支历史模型, 包括 1 位的预测器以及 10 位的分支 PC, 用于访问私有的历史模式或预测表。请问每种预测器的预测错误率分别是多少?

- 3.18 在图 3-46 的循环中, $A0, A1, A2$ 和 $A3$ 是基本块, b 是一个二进制变量, 在循环的连续迭代中按如下方式取值 (b_0, b_1, b_2)*, 即 b 的值分别为: $b_0, b_1, b_2, b_0, b_1, b_2, b_0, b_1, b_2, \dots, R$

是用作循环索引的寄存器。不幸的是， b_0, b_1, b_2 中的值是依赖于输入数据的，因此，当使用不同的输入数据运行代码的时候， b_0, b_1, b_2 的预期值可能是 8 种组合的任一种，并且概率相同。

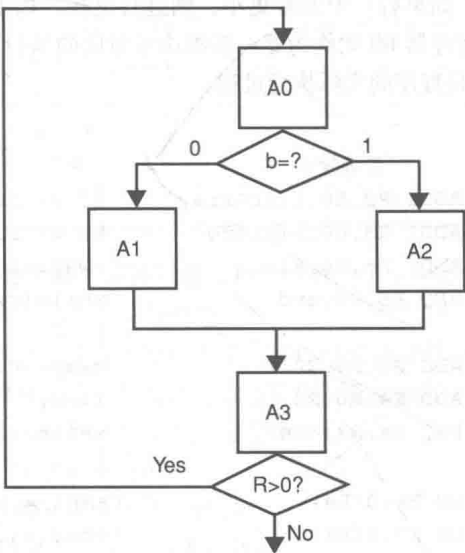


图 3-46 循环

回答下列问题，请忽略进入循环和退出循环的影响，假定分支预测缓冲器中的相关结构都已经完成预热。换句话说，假设循环迭代的次数非常大（无穷大）。

(a) 假设采用简单的一位预测器（没有历史记录），并假设在两个分支 PC 之间没有别名。对于 b_0, b_1, b_2 的每种可能值，给出 1 位预测器的预测错误率。

b_0, b_1, b_2	000	001	010	011	100	101	110	111
预测错误率 (%)								

(b) 假定采用简单的两位预测器（饱和计数器，无历史记录）重复上述问题，同样假设在两个分支之间没有别名。

b_0, b_1, b_2	000	001	010	011	100	101	110	111
预测错误率 (%)								

假设所有的模式出现的可能性相同，请问平均预测错误率是多少？

(c) 假定采用带有两位全局历史的两位预测器，重复上述问题。

b_0, b_1, b_2	000	001	010	011	100	101	110	111
预测错误率 (%)								

假设所有的模式出现的可能性相同，请问平均预测错误率是多少？

(d) 一位和两位预测器的问题在于它们忽略分支历史，所以它们无法记忆复杂的模式。请给出一个预测器设计，无论 b_0, b_1, b_2 取什么值都能使针对 b 的分支预测准确率达到 100%。在回答本问题时，请画图说明预测器结构，给出具体的访问方式，并给出所用到的所有表的大小。

3. 19 本题探讨分支和控制流变化对标量流水线结构下程序性能的影响。当取指和分支确定（或条件和目标确定）之间的流水级数越多时，分支的开销就越大。流水执行的上述效应推动了对分支预测的需求。本题主要探讨静态分支预测，基础机器为五级流水线。分支预测跳转和不跳转按照习

题 3.7 描述的方式进行处理。无条件分支在 ID 流水级执行。

本题将使用大家熟知的冒泡排序程序。在冒泡排序中，待排序元素将被反复扫描，每个元素与后一个元素进行比较，如果后一个元素更小，则进行交换。如果在扫描过程中没有元素交换，则排序过程结束。假设寄存器 R0 始终为零。基本块号对应的执行 trace 在代码段后面给出，假定下面的执行 trace 对应的是程序的执行过程。

示例代码：冒泡排序

基本块行号	标签	汇编指令	注释
1 1	main:	ADDI R2,R0,ListArray	R2 <- ListArray;
1 2		ADDI R3,R0,Listend	R3 <- Listend;
1 3		ADDI R5,R0,#1	swap<-1;
2 4	loop1:	BEQ R5,R0,end	while (swap!=0)
3 5		ADD R5,R0,R0	{
3 6		ADD R4,R0,R2	swap<-0;
4 7	loop2:	BEQ R4,R3,cont	i<-0;
5 8		LW R6,0(R4)	while (i < Listend)
5 9		LW R7,4(R4)	{
5 10		SLT R8,R7,R6	temp1 = ListArray[i];
5 11		BEQZ R8,skip	temp2 = ListArray[i+1];
6 12		ADDI R5,R0,#1	if (temp1 > temp2)
6 13		SW R7,0(R4)	{
6 14		SW R6,4(R4)	swap<-1;
7 15	skip:	ADDI R4,R4,#4	ListArray[i] <- temp2;
7 16		J loop2	ListArray[i+1] <- temp1;
8 17	cont:	J loop1	}
9 18	end:	JR R31	i++;
			}
			}
			return

上述代码中总共有 9 个基本块，执行 trace 对应如下基本块号：1 2 3 4 5 6 7 4 5 7 4 5 7 4 5 6 7 4 8 2 3 4 5 7 4 5 7 4 5 6 7 4 5 6 7 4 8 2 9

- (a) 请解释每个基本块 (1~9) 是如何确定的。
- (b) 分支行为和统计。填写如下的分支执行表 (表 3-33)，用 N 表示不跳转，T 表示跳转。该表格用来记录每个分支的执行模式，根据上面的执行 trace 进行填写。

表 3-33 分支行为

分支指令编号	分支指令执行编号									
	1	2	3	4	5	6	7	8	9	10
4										
7										
11										
16										
17										
18										

根据表 3-33，计算出下面表 3-34 所要求的统计信息。按照上面问题描述中给出的假设，无条件分支和有条件分支的平均代价分别是多少 (条件分支总是由硬件预测为不跳转，无条件分支在 ID 阶段执行)? 平均开销指的是每个分支平均损失的时钟周期数。

表 3-34 分支执行统计

分支指令编号	执行次数	跳转次数	不跳转次数	跳转百分比	不跳转百分比
4					
7					
11					
16					
17					
18					

执行该 trace 总共需要多少个时钟周期（包括所有的流水线填充和排空周期）？由于控制相关导致的阻塞总共损失了多少个时钟周期？

(c) 静态分支预测是影响分支执行的软件方法，可以用于减少控制相关引起的阻塞。分支操作码中带有静态预测位，表示执行该分支过程中可能的跳转方向。静态分支预测在五级流水线的译码阶段进行。这里引入如下新的分支操作码：

- BEQT-相等则跳转，静态预测为跳转；
- BEQN-相等则跳转，静态预测为不跳转；
- BNEZT-不等于 0 则跳转，静态预测为跳转；
- BNEZN-不等于 0 则跳转，静态预测为不跳转。

我们也可以定义其他类似的分支指令。静态分支预测在译码阶段进行，当静态预测跳转的分支指令进行译码时，机器也总是预测为跳转。相反，当静态预测不跳转的分支指令进行译码时，机器也总是预测为不跳转。

请使用新的带静态分支预测信息的操作码重写在代码序列中的所有条件分支指令（4，7，11）。使用表 3-34 中所生成的分支执行统计数据来确定操作码。

使用该 trace 重新执行新的代码，包括流水线填充和排空的周期，请问总的周期数是多少？IPC 又是多少？

3.20 为了改善题 3.11 中的超标量处理器性能，编译器可以应用局部（基本块内）或全局（跨基本块）调度。代码如下：

```
LOOP      L.D  F2,0(R1)
          ADD.D F4,F2,F4
          L.D  F6,-8(R1)
          ADD.D F8,F6,F4
          S.D  F8,0(R1)
          SUBI R1,R1,#16
          BNEZ R1, LOOP
```

- (a) 首先，编译器可以通过简单的代码移动来调度循环体内的代码，以尽量减少阻塞（局部调度）。请给出对该代码的局部调度方案，并计算在超标量架构上相比原本代码的加速比。
- (b) 接下来，编译器会尝试展开循环。不幸的是，寄存器 F4 存在跨迭代的依赖（一个迭代的依赖距离）。虽然编译器可以（也应该）通过对 F4 的重命名来避免代码移动中产生的 WAW 和 WAR 冲突，但寄存器重命名还是解决不了 RAW 冲突。不过我们依然可以获得一些额外的加速。请将循环展开两次，并进行指令调度。优化后相比于原代码的加速比是多少？编译器能够通过展开更多循环而获得更高的加速比吗？为什么？
- (c) 接下来尝试软件流水。请注意软件流水必须谨慎使用——流水循环的某次迭代中产生的值不能在下次迭代后使用，因为在下次迭代中值已经被重写过了。请对上述代码进行流水化，以获得最佳的加速比。软件流水比循环展开的效果更好吗？相比原代码的加速比是多少？

3.21 考虑如下程序，它搜索一块内存区域，并计算内存字和某个关键字匹配的次数：

```
SEARCH:    LW R5,0(R3)           /I1 读取数据项
           SUBI R6,R5,R2         /I2 和关键字比较
           BNEZ R6,NOMATCH       /I3 检测是否匹配
           ADDI R1,R1,#1         /I4 计算匹配次数
NOMATCH:   ADDI R3,R3,#4         /I5 下一项
           BNE R4,R3,SEARCH      /I6 直到所有项检测完成
```

- (a) 请识别出这段代码中的基本块，并绘制基本块的流程图。
- (b) 调到 NOMATCH 的分支之后的短基本块通常称为 *hammock*，循环中的 *hammock* 是乱序执行处理器效率不高的重要原因。为了便于理解，我们可以针对循环的前两次迭代使用支持推测的 Tomasulo 算法（推测寄存器的值存储在 ROB 中）填写表 3-29 所示的执行表。循环内的分支（BNEZ）在这两个迭代中都预测为跳转，不过实际执行时，它在第一个迭代中是不跳转的（匹配），但在第二个迭代中是跳转的（不匹配）。由于分支在第一个迭代中预测错误，执行过程必须回滚。假定，预测错误的分支到达 ROB 顶端时进行处理。在分支达到 ROB 顶部的下一个周期，ROB 和流水线后端都被刷掉，并重置 RAT 表。请给出在执行表中的所有活动。第一次迭代后，循环底部的分支被正确地预测为跳转。
- (c) 为了解决上述问题，建议在 ISA 中增加一条新的预测指令以消除 *hammock*。新加的指令如下：

```
CMOVZ R1,R2,R3    /R1 <- R2 if R3=0
```

使用上面的新指令重新编写循环代码来去掉内部分支和 *hammock*。确保修改后代码与原有代码得到的结果相同，此外，不会触发不该发生的异常。请解释为什么新代码没有原代码的性能问题。

- (d) 在类似于表 3-29 假设的乱序执行处理器中，实现谓词指令（条件指令）是非常具有挑战性的。一种可能的实现是等到 R3 的值已知时才对 CMOVZ 进行分发。然而，这种方法效率不高，就好像 *hammock* 中的分支没有推测执行一样，因此效果也类似于非推测的 Tomasulo 算法。另一方面，当谓词寄存器的值还未确定时就调度谓词指令会很麻烦。请解释一下会有什么问題，并提出解决方案。

3.22 本题主要和图 3-47 所示的五级流水线的 VLIW 扩展相关。各级流水线之间的流水线寄存器在图中没有画出，但都存在并且按和以前一样的方式进行命名，如 ID1/EX1。条件分支和无条件跳转都带一个长指令的延迟，并且都是在 ID4 阶段执行，这样跟在分支指令后的长指令在取指阶段总能执行，无论分支是否跳转。

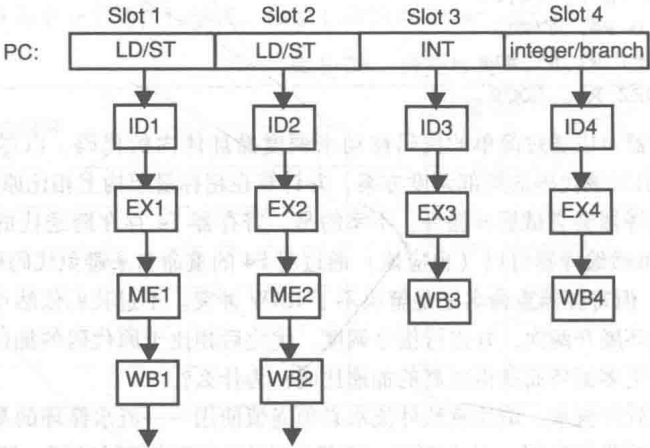


图 3-47 VLIW 整型流水线

- (a) 基于以下三种不同情况分别计算不同操作的延迟。

三种情况是：

- 完全不支持前递。
- 只支持内部寄存器的前递。
- 支持完全前递（包括内部寄存器之间的前递）。

需计算的操作延迟如下：

- load 到整型操作（在寄存器）。
- load 到 store（在内存操作数和地址寄存器）。
- 整型操作到 load 或者 store（在地址寄存器）。
- 整型操作到分支（在寄存器）。
- load 到分支（在寄存器）。

这里所讲的完全前递，是指所有的前递路径，前递都只是从流水线寄存器到 EX 阶段的输入。在完全前递情况下计算指令到分支的操作延迟时，这一点非常重要。

(b) 再考虑如下统计在值向量和某个关键字匹配次数的代码：

```
SEARCH:    LW R5,0(R3)           /I1 读取数据项
           SUB R6,R5,R2          /I2 和关键字比较
           BNEZ R6,NOMATCH       /I3 检测匹配
           ADDI R1,R1,#1         /I4 计算匹配次数
NOMATCH:   ADDI R3,R3,#4         /I5 下个数据项
           BNE R4,R3,SEARCH      /I6 直到所有项检测完成
```

由于循环中的分支影响了代码的并行化，所以我们将其替换为条件移动指令 CMOVZ：

```
SEARCH:    LW R5,0(R3)           /I1 读取数据项
           SUB R6,R5,R2          /I2 和关键字比较
           ADDI R7,R1,#1         /I3 假设匹配
           CMOVZ R1,R7,R6        /I4 如果匹配，R1加1
           ADDI R3,R3,#4         /I5 下个数据项
           BNE R4,R3,SEARCH      /I6 直到所有项检测完成
```

为了进一步提高并行性，编译器将循环展开三次。针对上面的三种不同前递机制，请分别给出性能最好的 VLIW 代码实现。使用和表 3-19 相同的格式。

(c) 基于针对三次循环展开后的调度，请给出寄存器数目限制下三种不同前递机制对应的原始循环体执行时间计算公式。

3. 23 计算前缀和的操作输入是一个向量 $X = (x_1, x_2, x_3, \cdots, x_n)$ ，输出向量 $Y = (y_1, y_2, y_3, \cdots, y_n)$ ，计算公式如下：

$$y_i = \sum_{j=1}^i x_j \quad (i = 1, \cdots, n)$$

下面是计算前缀和对应的一个简单循环：

```
      L.D F0,0(R1)
LOOP:  L.D F2,-8(R1)
      ADD.D F0,F2,F0
      S.D F0,-8(R1)/      O3
      SUBI R1,R1,#8
      BNEZ R1,R2 LOOP
```

R2 的值等于 R1 最后的值减 8，使用如图 3-27 和表 3-17 所示的结构，分支始终预测为跳转，L1 数据 cache 始终命中。

(a) 首先，我们在 load/store 队列中采用非常保守的策略：load 操作需要等到和之前 store 的所有可能冲突都确定时，才能发射到 cache。如果 load 依赖于之前的 store，那么 load 需要一直等

待直到 store 更新完 cache (不支持 store 到 load 的前递)。

(b) 其次, 我们假设仍然采用同样的保守策略, 但是支持 store 到 load 的前递: store 结果一旦确定就会前递给 load。

(c) 最后, 在乐观策略下重复上面的 (b)。

3.24 假设我们对表 3-3 中的每条指令做一个增加谓词的扩展, 以 ADD 指令为例:

```
ADD R1, R2, R3          /* R1 <- R2+R3
```

这个 ADD 可以描述为:

```
(R4)   ADD R1, R2, R3    /* R1 <- R2+R3 if R4!=0
```

```
/* NOOP if R4 = 0
```

```
(~R4)  ADD R1, R2, R3    /* R1 <- R2+R3 if R4 = 0
```

```
/* NOOP if R4!= 0
```

同理, load 指令描述成:

```
(R4)   L.W R1, 0(R2)     /* R1 <- Mem[(R2)+0] if R4!=0
```

```
/* NOOP if R4 = 0
```

R4 是谓词寄存器, 其值不限定于 0 或 1。注意, 谓词字段只包含一个寄存器 (没有表达式)。每条指令现在都有两个版本: 有谓词的版本和没有谓词的版本。当然, 此时目标代码的格式也不同, 可能需要更多的位来表示 (可能需要增加 5 位来指定谓词寄存器)。但是, 这不是本题所关心的问题。当没有谓词寄存器时, 表示该指令不是谓词版本。

考虑如下代码 (A, B, C 和 D 是内存中的 32 位整型字):

```
if (A>=C&&A>=B) A:= B+C;
    else if (B<=D | A==C+D) B:=A-C;
```

为了简化起见, 假设 A, B, C 和 D 的绝对地址分别存储在 R1, R2, R3 和 R4 中。下面是使用了分支的基本指令集的一种可能代码序列:

```
I1      LW R5,0(R1)      /加载A
I2      LW R7,0(R3)      /加载C
I3      SLT R8,R5,R7     /判断A<C
I4      LW R6,0(R2)      /加载B
I5      BNEZ R8, else
I6      SLT R9,R5,R6
I7      BNEZ R9, else     /判断A<B
I8      ADD R10,R6,R7
I9      SW R10,0(R1)
I10     J exit
I11     else LW R11,0(R4)  /加载D
I12     SLT R12,R11,R6
I13     BEZ R12,else1     /判断D<B
I14     ADD R13,R7,R11
I15     BNE R5,R13,exit   /判断A==C+D
I16     else1 SUB R14,R5,R7
I17     SW R14,0(R2)
I18     exit
```

使用谓词和/或非谓词指令将上述代码翻译成新的汇编代码, 不允许使用条件分支指令或无条件跳转指令。

请确保新的代码不会引起本不应该出现的异常。

3.25 本题分析题 3.24 中使用分支和跳转的代码版本。

答: (a) 请识别出代码中的所有基本块。

- (b) 请在习题 3.22 所描述的 VLIW 机器上对代码进行调度，只能使用局部优化（即基本块内优化）。注意，分支延迟一个周期，但是习题 3.24 的代码中没有针对整个延迟的调度。
- (c) 某个学生在计算机科学类项目中设计了一个简单的编译器，我们用该编译器对代码进行了全局重新组织，如下所示。该学生认为在获得相同结果的前提下，这个版本可以获得更好的性能（这段代码中也没有考虑分支延迟的问题）：

```
I1          LW R5,0(R1)          /加载A
I2          LW R7,0(R3)          /加载C
I3          SLT R8,R5,R7
I11         LW R11,0(R4)         /加载D
I12         SLT R12,R11,R6
I14         ADD R13,R7,R11
I5          LW R6,0(R2)          /加载B
I6          SLT R9,R5,R6
I8          ADD R10,R6,R7
I16         SUB R14,R5,R7
I4          BNEZ R8, then        /判断A<C
I7          BNEZ R9, then        /判断A<B
I9          SW R10,0(R1)         /执行if子句
I10         J exit
I13         then BNEZ R12,then1    /判断D<B
I15         BNE R5,R13,exit       /判断A==C+D
I17         then1 SW R14,0(R2)    /执行then子句
I18         exit
```

这个学生认为上述代码是正确的，因为内存中的结果总是和原始代码的结果一致，然而，这个学生还忽略了一些东西：

- LW 指令跨越 store 上移了，这会引起内存冲突；
- 导致异常的指令跨越跳转和分支指令上移了，这会引起控制冲突，并且触发本不应该出现的异常。

不过还好，这个学生没有跨越分支上移 store 指令。（为什么说这是一件好事？）

任何情况下，都需要一套机制用于检测并纠正内存冲突和异常引起的问题。因此，我们使用和 IA-64 ISA 中类似的机制和指令。

首先来看异常的处理。先假设除了 load 和 store 外，其他指令不会触发异常（请记住，store 永远不能推测执行）。当 load（及其相关指令）提升到分支指令前时，load 就变成了操作码为 LW.s（如 LW.s R1, 0(R2)）的推测 load 指令。由于现在 load 是推测的，它本来可能不需要执行，因此暂时不会发出对程序可见的异常信号。例如，缺页异常是程序不可见的，但地址不对齐则是程序可见的。当推测 load 引起这种对程序可见的异常时，load 指令通常返回一个未定义值，从而将目标寄存器标记为有毒并将其内容替换为一个异常描述符。在原始代码中的 LW 位置，插入格式为 check.s R1, repair 的检查指令。如果推测 load 及其相关指令本不应该执行，那么 check.s 指令也不执行。与之相反，如果推测载入指令本应该执行的话，那么 check.s 指令也执行并查找寄存器。如果寄存器有效，一切照常并继续执行。如果发现寄存器处于有毒状态，则跳转到修复代码执行，修复代码本质上是在非推测状态下重新执行被前移的指令序列。这次执行时将会接受异常。

然后我们使用类似机制处理内存冲突。当 load 及其相关指令跨越一个或多个 store 操作前移，并且编译器无法消除内存地址歧义时，该 load 值变成推测的，同时 load 操作变成 LW.a（例如，LW.a R1, 0(R2)）。在原始代码的 LW 位置，插入格式为 check.a 0(R2)，repair 的检查指令。该指令需要结合被称为先进 load 地址表（ALAT）的小型硬件表一起工作：LW.a 将其地址插入

ALAT 中, 如果具有相同地址的 store 操作在 LW. a 和 check. a 之间执行, 那么将该地址从 ALAT 删除, 后续 check. a 可以检测到这一点。如果由 LW. a 返回的值是旧的, 那么 check. a 将跳转到修复代码, 主要是重新执行 load 操作及其相关指令。

请完成下列操作:

- 在新的代码中添加指令以检查错误推测 (包括内存操作和异常);
- 对新代码进行局部调度, 利用上分支和跳转延迟槽的特点;
- 基于习题 3.22 的 VLIW 机器进行代码的调度。

在所有可能的情况下, 在 VLIW 机器上的这份新代码能获得的加速比是多少? 支持推测 load 操作的新代码在什么时候性能更好?

- 3.26 根据 Amdahl 定律, 向量处理器还需要一个快速的标量处理器来提高那些无法向量化代码的执行速度。一个常见的向量操作是两个向量的点积, 其结果是一个标量, 这种点积操作是矩阵乘法和大多数信号滤波中的基本操作。两个维数为 n 的向量 X 和 Y 的点积由下式给出

$$X \cdot Y = \sum_{k=1}^n x_k y_k$$

对应的 C 语言代码如下:

```
for(k=0; k<n; k++) p += x[k]*y[k];
```

这段代码的问题在于它存在跨循环的依赖。不过, 在有高性能标量处理器做支撑的向量处理器中, 这个算法的计算效率依然很高。点积中主要有两种操作: 一个是两个向量相乘, 然后是结果的累加。

在执行该代码时, 循环划分成 64 个元素的片段进行分段开采, 每两个输入向量片段依次相乘, 然后将这些相乘结果存储到结果向量寄存器。处理完所有片段后, 将结果向量寄存器的 64 个元素加在一起形成点积结果。为了加快每个片段的处理, 我们将向量操作进行链接后并行计算。

- (a) 假设硬件结构与图 3-41 所示的类似, 总共有 8 个向量寄存器, 每个有 64 个元素, 假定存储体数量很大 (例如 1024), 这样存储体之间不存在冲突。访问存储体的时间是 30 个周期, 步长为 1, 乘法流水线的延迟是 10 个周期, 加法流水线的延迟是 5 个周期。

请给出使用如 3.7 节所示的向量 load 和算术指令处理每个 64 元素片段的代码, 并计算点积操作所花费的时间, 假定每个向量大小为 1024 (忽略最后对元素进行累加的标量处理阶段)。

- (b) 使用相同的算法计算 1024×1024 的矩阵乘法需要多少个时钟周期 (忽略最后的标量计算阶段)。

- (c) 将向量循环展开两次 (假设有足够的向量寄存器), 给出计算点积的代码, 并计算 1024×1024 矩阵乘法 (忽略最后的标量计算阶段) 所需的时钟周期数。

- 3.27 之前我们一直假设向量处理器配备的是一个简单的、交叉访问的存储系统, 总共有 4 个存储体, 存取延迟为 4 个周期, 访问步长为 1。不过, 在实际中, 很可能会有更多的存储体, 而且访问复杂结构的步长也可能会有所不同。

- (a) 现在假设有 32 个存储体, 每个存储体的存取时间为 8 个时钟周期, 一次只能同时进行一个向量 load 或一个向量 store。在采用如图 3-40 所示的简单交叉存取方案, 可避免冲突的向量步长 (从 1 ~ 32) 是多少? 通过将步长对 32 取模, 可否将该结果类推到所有步长的情况?

- (b) 假设有 31 个存储体, 每个存储体的存取时间为 8 个时钟周期, 一次只能进行一个向量 load 或一个向量 store。在采用如图 3-40 所示的简单交叉存取方案的前提下, 可避免冲突的向量步长 (从 1 ~ 31) 是多少? 通过将步长对 31 取模, 可否将该结果类推到所有步长的情况?

- (c) 若使用 31 个存储体而不是 32 个, 有什么优点和缺点?

- 3.28 并非所有向量机都是 load/store 类型的。事实上, 第一台向量机——CDC Star-100 (由现已解散的 CDC 公司建于 20 世纪 70 年代初) 就是一个纯粹的内存到内存模式的向量机。输入向量从内存直

接流入流水线单元，并将结果直接流回内存。这种风格的架构访存非常频繁，数据在内存中的来回移动导致了巨大的延迟。尽管在 20 世纪 80 年代初，CDC Star-100 进一步升级为 CDC Cyber-205，但是这种类型的机器最终还是被 Cray 公司的机器（load/store 向量机）所取代（从 1976 年的 Cray-1 开始）。在 Cyber-205 机器中，从内存取的向量元素允许有 64K 个之多。内存到内存的向量机不能像习题 3.26 那样对向量寄存器中的点积部分结果进行累加。相反，它们要在输入操作数从内存流到向量单元的“传输过程中”完成点积计算。因为点积是非常重要的操作，所以可以为它专门设计一个特殊的功能单元，如图 3-48 所示。

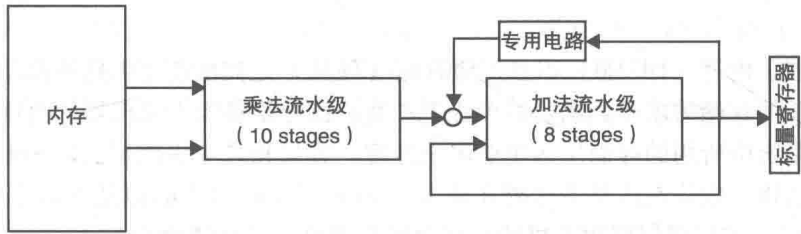


图 3-48 内存到内存向量机上的点积计算功能单元

- (a) 使用图 3-48 中的功能架构，说明如何计算两个 64K 向量的点积（从开始到结束，包括在专用电路中的行为）。描述执行的各个阶段并解释应该如何逐个时钟周期地对流水线进行控制。
- (b) 完成计算的电路（图中的 circuit）非常简单。请描述该模块主要用于做什么，并绘制出可能的硬件结构框图。
- (c) 假设内存访问时间为 100 个周期，并考虑在特殊电路上花费的时钟周期，再加上存储到目标寄存器所花费的时钟周期，那么计算两个长度为 64K 向量的点积总共需要多少个时钟周期？假设访问交叉存储时没有冲突。
- (d) 通常我们用 N 来表示内存向量的长度。那么执行长度为 N 的两个向量的点积需要多少个周期？如果想要至少达到 64K 向量点积性能的一半以上，那么最小的向量长度是多少？
- (e) 对比本题的向量实现方式和 3.3.2 节所示的静态流水线方式。假定现在有 3 个并行流水线，一个用于 load/store/分支/整型，一个用于浮点加法，一个用于浮点乘法。不过考虑到习题 3.25 代码中的跨循环依赖，我们无法通过循环展开获得太多的加速。因此，在最好的情况下（假定 cache 都命中），静态流水线的最小延迟大约是

$$3(\text{loads}) + 10(\text{multiply}) + 8(\text{add}) + 2(\text{address} + \text{branch}) = 23 \text{ 时钟周期 / 循环迭代}$$

基于上面所提供的数据，为了使向量实现方式一定快于标量机，最小的向量长度 N 应该是多少？

存储层次

4.1 概述

随着处理器、内存（DRAM）以及二级存储（硬盘）之间的速度差距不断增大，在保证应用程序不断增长的存储需求得到满足的前提下，我们已经很难跟上处理器的速度来提供指令和数据了。现代系统中所用的存储层次主要考虑速度、容量和成本等因素，图 4-1 给出了一种常见的存储层次结构，虚线左边是高速缓存层次（cache hierarchy），右边是虚拟存储层次（virtual memory hierarchy），虚拟存储层次中可能会包含硬盘缓存（图中未画出）。

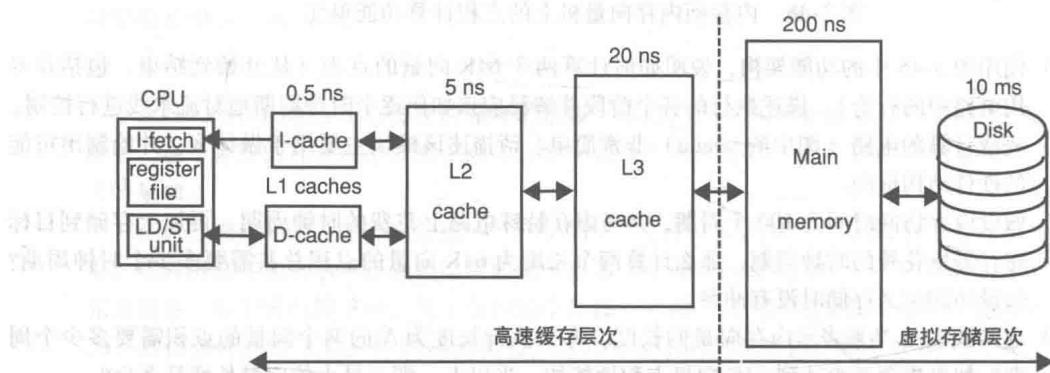


图 4-1 带有 3 级高速缓存的典型存储层次

近年来，人们发现处理器的速度（主频几 GHz，一个时钟周期可以执行多条指令）同主存的速度（一次访问需要几十到几百 ns）之间的差距在呈指数增长，这一问题通常称为存储墙问题。不同容量和访问速度的多级 cache 组成的层次结构就是用来缩小这个差距的。此外，每一级的 cache 也变得越来越复杂，以减少 cache miss 时的开销。如今的动态乱序执行处理器在任何时刻都可能存在十多个待处理的访存操作，为了支持这一场景，现在的非阻塞（lockup-free，或者叫 non-blocking）cache 可以在同一时刻处理多个 cache 命中和未命中的情况。同时，指令和数据也会在真正被用到之前就被预取到 cache 中。在本章，我们将讨论这些针对 cache 设计的优化。

在多处理器和多核系统中，存储墙的问题更加严重，因为存在对共享资源的竞争、一致性的维护，以及互连网络引起的延迟等。因此，人们尝试开发对存储一致性模型（将在第 7 章中介绍）的放松以及内存推测技术等，以便尽可能多地隐藏内存访问的开销，从而减少其对处理器执行速度的影响。在本章，我们只考虑单核系统的存储层次。

cache 是由 cache 行组成的，任何时刻，一个 cache 行可以保存一个同它大小相同的内存块。由于 cache 比主存小很多，多个内存块会共享一个 cache 行，在不同时刻可能保存不同的内存块。内存块和 cache 行之间的映射关系是 cache 的重要特征。当处理器在某一级的 cache 中找到访问的地址单元时，我们称为 cache 命中，当处理器无法找到访问的地址单元时，称为 cache 失效。

主存和硬盘之间的速度差比处理器和主存之间的速度差要大得多，这是因为内存以电子设

备速度在工作 (ns 级), 而磁盘以机械运动速度在工作 (ms 级)。磁盘空间被划分为两部分: 文件系统区和虚拟内存系统区。文件系统区保存了存储的代码和数据, 文件可以被应用程序显式地打开、读写以及关闭。磁盘上的虚拟内存系统区作为主存的备份, 就好像主存是磁盘上虚存空间的 cache。虚存通常由操作系统以及内核以软件方式管理, 因此虚存管理是操作系统层面的问题, 并不是体系结构层面的问题。在本书中, 我们主要关注体系结构对虚存的支持以及同虚存软件之间的硬件交互。

主存按照物理页帧进行组织。任意时刻, 一个物理页帧可以保存同样大小的一个页 (page)。进程可以访问的虚存空间也被组织成页的形式, 所有的活动页在磁盘上都有备份, 并且在某一时刻可能在主存的页帧中。当处理器访问的地址单元在主存中时, 称为页命中, 无需软件参与就可以将该单元内容返回给处理器。相反, 当地址单元不在主存中时称为缺页, 需要内核进程将页调入主存页帧中。

在本章中, 我们将介绍以下内容:

- 存储层次的概念, 访存局部性原理, 存储层次中的一致性问题, cache 和内存包含, 这些将在 4.2 节中讲解。
- cache 层次, cache 映射和访问, 替换和写策略, cache 失效的分类, 高性能 cache (非阻塞 cache, cache 预取和预加载), 这些内容将在 4.3 节中讲解。
- 虚存的硬件支持, 页表和旁路转换缓冲, 虚地址 cache, 这些将在 4.4 节中讲解。

4.2 金字塔形存储层次

图 4-2 中给出了一个存储层次的示意图, 金字塔的形状反映了不同层次存储的容量。CPU 及寄存器组在金字塔顶端, 作为第 0 层, 大容量存储器 (硬盘) 在金字塔底端。寄存器组容量小, 集成在处理器中, 并且为了提高速度进行优化, 可以在 1 或 2 个周期内完成访问。寄存器内容被编译器离线管理, 静态调度从内存加载或存储到内存中。寄存器并不看作真正的存储器, 因为处理器通过寄存器号而不是通过内存地址访问它们。

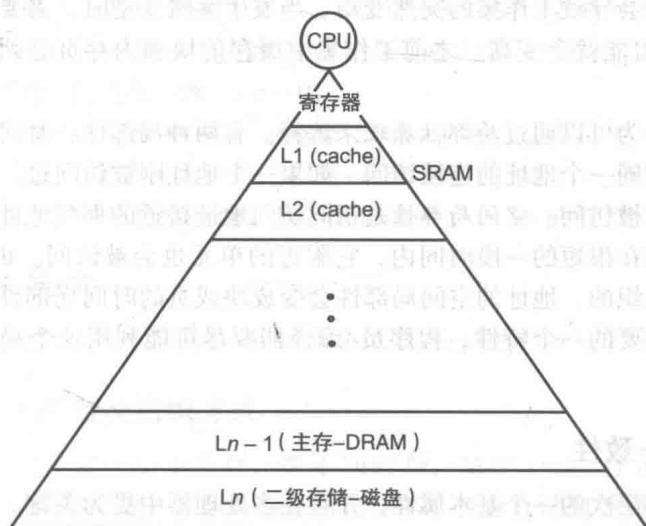


图 4-2 金字塔形存储层次

从金字塔顶端往底端看, 中间的几级 cache 缩短了处理器和内存之间的速度差。cache 是动态加载处理器访问内存单元的高速缓存。一般一级 cache (L1) 和二级 cache (L2) 是片上的, 其他层级的 cache 是片外的。为了支持数据访问和取指令的重叠, L1 cache 通常会拆分成独立

的指令 cache 和数据 cache (这种结构也称为“哈佛 cache”), 而低层次的 cache 则一般是混合 cache (既包括指令也包括数据, 这种结构也称作“普林斯顿 cache”)。L1 和 L2 cache 一般由高速 SRAM 阵列组成, L3 和 L4 cache 则可能是由 DRAM 阵列组成。在这种分层 cache 结构中, 通常 cache 容量越大, 就越有可能包含所需访问的地址单元, 但是访问速度也会越慢。大容量 cache 之所以速度较慢是因为访问时间主要取决于连线延迟 (地址线、行选线、列选线), 这些连线延迟很难随着技术进步而缩短。此外, 大容量的存储器需要更大的译码器和多选器, 这也导致速度变慢。所有层次的 cache 存储器都是由硬件直接管理的。

注意我们对 cache 层级的编号: 第 1 级在最上边, 沿着金字塔向下, 级数编号逐渐增加。此外, 我们还将使用“上一级”或“下一级”表示在金字塔中的位置。在最后一级 cache 的下面是主存, 它包含了处理器所要访问的所有指令和数据。

金字塔的底部一级的管理与 cache 不同, 主存和二级存储之间的虚存管理问题要追随到 20 世纪 70 年代虚存的提出。为了使 CPU 在执行磁盘 I/O 操作时也尽可能忙碌, 分时操作系统支持将处理器在多个进程之间进行分时复用。CPU 进程的调度和虚存的管理交由操作系统的内核程序负责。

4.2.1 访存局部性

cache 和虚存之所以能够成功, 是由于通常程序执行过程中都有访存局部性这一特点。如果程序访存是随机的, 那么 cache 和主存的效率将难以接受, 这样缓存和虚存也就不会有效。幸运的是, 在任意一段时间内程序通常都只会访问其内存空间中的一小部分。例如, 一个进程在主程序中开始, 然后跳到了不同的程序和代码段, 这些程序和代码段都访问自己的数据集。每个程序中都会包括一些循环反复访问自己的代码和数据。因此, 在任意时刻, 一个进程访问的内存单元 (也称作工作集) 虽然随着时间一直在变化, 但是通常会限定在某个代码段、程序段或循环中。在任意时刻, 处理器只会访问工作集中的地址, 进程执行其代码的某部分, 由于这部分代码处在当前工作集中, cache 和主存的失效率就会相对较低。当进程从代码的一部分转而执行到另一部分时, 会导致工作集的突然变动。当发生突然变动时, 需要访问新的存储区, cache 和内存的失效率可能就会变高。之前工作集中缓存的块和内存页必须被替换掉, 从而给新的工作集留出空间。

程序的这种典型访存行为可以通过局部性原理来解释。有两种局部性: 时间局部性和空间局部性。时间局部性是指对同一个地址的连续访问: 如果一个地址刚被访问过, 那么很可能在很短的一段时间内它会再次被访问。空间局部性是指同访问地址接近的那些地址单元: 如果一个地址被访问, 那么很可能在很短的一段时间内, 它附近的单元也会被访问。由于 cache 和主存都是按照连续的块进行组织的, 地址的空间局部性会变成块或页的时间局部性。因此, 访问局部性是代码片段中非常重要的一个特性, 程序员/编译器要尽可能利用这个局部性来提高访存速度。

4.2.2 存储层次中的一致性

存储一致性是所有存储层次的一个基本属性, 并且在多处理器中更为关键。不过, 考虑到单核中也有很多对 cache 层次的优化, 因此在单核系统中, 一致性问题也是一个值得关注的问题。任何对单核系统的 cache 优化方法都应以确保一致性为前提。“一致性”并不是说在所有时刻同一个存储地址的拷贝都要完全一样, 就好像整个存储系统是完全统一的。

在单核系统中, 指令可能会乱序执行或推测执行, cache 可能是写穿透的或者是写回并且无阻塞的, 但是我们必须保证最后的执行结果同按照指令序一条一条执行后的结果是完全一样

的。具体来讲,无论存储层次怎样,一个 load 必须总是返回对同一个地址上一次 store 的值(按照程序序)。这就是单核系统中一致性的定义。

当存储层次更新时,为了维护一致性,这个更新必须要传播到低级存储中。如果 cache 是写穿透的,所有对 cache 的更新都要传播到低级存储中,如果 cache 是写回的,则当 cache 行中的块被替换掉时,需要将更新后的内容写回下一级。

4.2.3 存储包含

由于处理器只能访问主存中的地址单元,任何一个被访问的指令或数据都必须在主存中。因此任何一级 cache 中的块在主存中都要有备份,即 cache 中的任何内容都要包含在主存中。而 cache 是否包含上一级 cache 的内容则不是必须的,可以进行设计选择。我们称 j 级 cache 包含 i 级 cache ($j > i$),如果满足以下条件:

- 任何缓存在 i 级 cache 中的内容也都缓存在 j 级 cache;
- j 级 cache 中的状态包含 i 级 cache 的状态。

第二点表示访问 i 级 cache 的权限要么同 j 级相同,要么比 j 级严格。例如,如果 i 级 cache 的备份是可读可写的,那么这个备份在 j 级也必须可读可写。如果备份在 j 级是只读的,那么在 i 级就不能是可写的。

cache 层级之间的包含关系是一个很有用的特性,尤其是在维护一致性方面。在单核系统中,包含关系有利于更新向低级别 cache 的传播。在多处理器系统中,如果 j 级 cache 中没有需要的数据,就不需要查看 i 级 cache 了。为了维护包含关系,每次 j 级 cache 的备份被替换时, i 级中相应的备份也都要被无效掉。此外,无论何时一个块被取到 i 级 cache,它必须也要被取到低于 i 级的所有 cache 中。包含关系的一个主要缺点是,处理器节点所能缓存的总空间受限于 cache 层次中较低级 cache 的容量大小。

为了防止缓存空间的浪费,有的 cache 层次之间采用互斥(exclusion)关系。如果保持 cache 互斥,当 $i < j$ 时, i 级 cache 中的块就不会出现在 j 级 cache 中。因此处理器节点总共的缓存空间就是节点中所有缓存空间之和。为了保证互斥,每当数据块取到 l 级 cache (发生 miss) 时,所有低于 l 级的 cache 中对应的该数据块都要被无效。此外,如果一个块从 l 级被换出,它可以分配到最接近 l 的更低一级 cache 中。

当然,我们也可以不维护 cache 的包含和互斥关系。这样,某一级内存块的分配和替换不会影响到其他级 cache。但是,除了简化控制 cache 的硬件之外,这种做法并不会带来其他明显的好处。在本书中,如果不特别强调,我们默认维护了 cache 的包含关系。

4.3 cache 层次

cache 行为主要的影响因素是 cache 大小以及内存块到 cache 行的映射方式。

4.3.1 cache 映射及组织方式

由于 cache 的容量远小于主存,在不同时刻,每个 cache 行必须可以存放多个内存块。cache 由两部分存储体组成:目录存储和数据存储。目录存储包括当前存储在每个 cache 行中的内存块标识(ID 或 tag)以及一些状态位。状态位中至少要有一个有效位(V, valid bit),有效位指明 cache 行中的数据是否有效,并且可以在某些时刻被置为无效状态。随着讲解的深入,每个目录项中会增加更多的状态位。数据存储中包括了内存块的备份。

内存块与 cache 行之间的映射是基于块地址的。物理内存地址被分为两部分:内存块地址以及块内偏移,如图 4-3 所示。在按字节寻址的内存中,cache 行内和内存块内的字节偏移是

相同的。

物理地址		
内存块地址		块内偏移
标识位	cache索引位	块内偏移

图 4-3 访问直接映射 cache 的地址域

cache 映射方式有三种：直接映射、组相联映射和全相联映射。

直接映射 cache

在直接映射 cache 中，对于一个给定的内存块总会映射到同一个 cache 行中，通过哈希 (hash) 块地址的方法就可以得到这个 cache 行的位置。尽管有很多哈希方法，但是最简单的一种方法是位选择哈希，这种方法通过块地址的某些位来选择 cache 行。由于指令和数据的访问有局部性，因此通常用块地址的最低几位来选择 cache 行，见图 4-3 中的 cache 索引位。剩余的几位（块地址的最高几位）形成存储在 cache 行中的内存块标识，并被存储在目录中。

图 4-4 给出了直接映射的 cache 架构以及访问方法。图 4-4a 给出的是窄 cache 的情况，数据存储的宽度是一个字。由于本例中 cache 行大小是两个字，因此数据存储的高度是目录存储的两倍。一般来讲，数据存储的高度与目录存储的高度比为 $W = 2^w$ ，其中， W 是一行中的字数。给定块大小为 $B = 2^b$ ，行数为 $S = 2^s$ ，每个物理地址的位数为 $N = 2^n$ ，则识别 cache 行的标识位为 $n-s-b$ 。下面给出了读访问所需的两步操作。

- cache 索引 (cache indexing)。通过块地址的最低 s 位，可以取到目录项，同时，可以通过块地址的最低 s 位以及块偏移的最高 w 位取到数据存储。cache 索引的速度与标准 SRAM 访问速度相同。
- 标识检查 (tag checking)。将索引到的 cache 行的目录标识与块地址的最高几位进行比较，并且也要检查状态位（比如有效位）。如果标识匹配并且状态位与访问行为一致，数据就可以被传递给下一级（cache 命中），否则会触发 cache 失效。

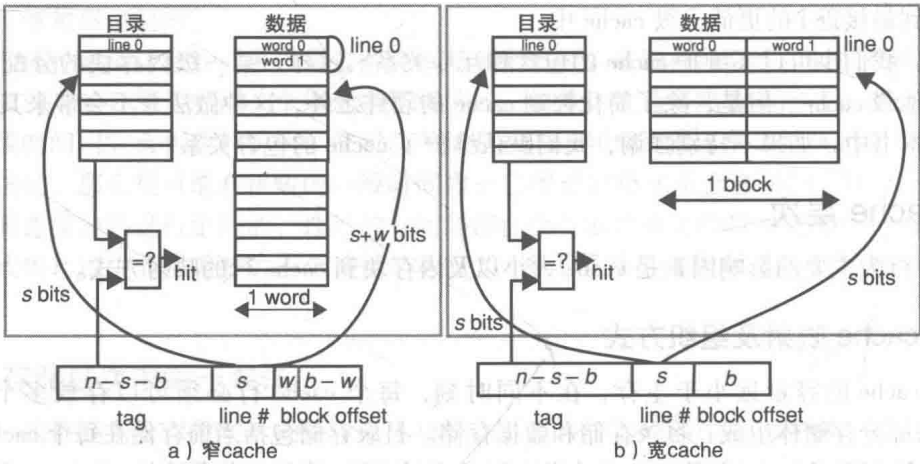


图 4-4 直接映射 cache 的索引

尽管 cache 中的数据存储行可以小于一个字，但实际设计中并不提倡这么做，因为一次 cache 命中需要访问不止一次数据存储才可以取出需要的字。图 4-4b 给出了宽 cache 的示意图：

数据存储的宽度是整个行，因此目录存储和数据存储的高度相同。目录和数据存储都可以通过块地址的低 s 位索引到。宽 cache 的主要优点是，当 cache 失效时，只需要一个数据存储访问周期就可以重新加载失效块，而在窄 cache 中，需要 W 个周期才可以完成重加载。由于块是访问一个 cache 行的最大单位，因此需要将 cache 的宽度设计成比整个行的大小宽一些。但是，为了减少设计宽 cache 的复杂度，需要在重载失效块的耗时与设计复杂度之间进行协调，选择适当的 cache 宽度。在这种情况下，用多少位作为块偏移取决于数据存储中一行有多少字。

组相联 cache

直接映射 cache 的最优点处是 cache 命中时的开销小，而它最大的缺点就是将块映射到固定的 cache 行。内存块中有很多块会被映射到同一个 cache 行，当多个内存块竞争一个 cache 行时会导致很高的失效率。为了缓解这个问题，又要保证 cache 的简单以及命中时的低开销，绝大多数 cache 都采用组相联的方式。组相联 cache 将几个 cache 行分为一组，对每组的映射是直接映射，但是一个块可以被放在组内的任何一个行中。

图 4-5 给出三路组相联 cache 的架构，也就是说，每个 cache 组包括 3 个 cache 行。组数为 $S=2^s$ ，所有相联的路就是图 4-4a 或 b（窄 cache 或宽 cache，或介于两者之间）中的 cache 片，每片按照图 4-4 中的方式索引。利用内存地址的 s 和 $s+w$ 位，同时分别（在窄 cache 片的情况下）取得 3 个目录项和数据存储块。接下来将每片的块地址标识与 3 个目录项进行比较，每片返回命中或失效，然后触发失效或在命中的片中选取需要的字。直接映射 cache 可以被看作一路组相联的 cache。

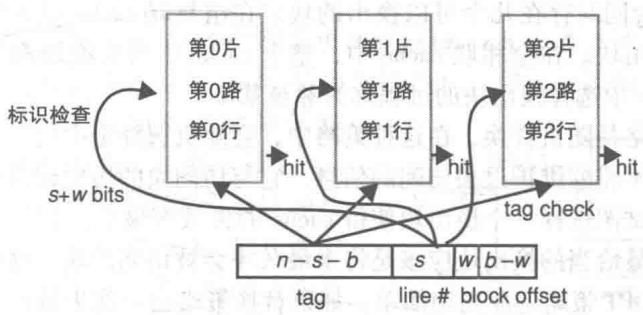


图 4-5 三路组相联 cache

对于组相联的读访问，目录和数据存储的索引是并行进行的，但是写访问则不同。在更新数据存储的同时不能同时读取目录项，因为在 cache 失效的情况下，这个 cache 行是不应该被修改的。因此写访问至少需要两个周期：一个周期检查标识来判断是否命中，一个周期进行写操作。当写操作是突发式的（burst）时，这两个周期可以流水化。

通常，一个 N 路组相联中会出现“热点组”，热点组是指超过 N 个内存访问产生竞争。典型的 cache 相联度在 2~8 之间，如果超过 8 路，并行访问各片以及判断命中/失效的逻辑部件会变得很慢并且十分复杂，并且对命中率几乎没有什么提高。为了避免热点组的出现，可以把 cache 做成全相联的，即组大小就是整个 cache 的大小。

全相联 cache

全相联 cache 的结构与组相联 cache 很不相同，如图 4-6 所示。在全相联 cache 中，一个内存块可以被映射到任何一个 cache 行，因此不需要通过若干位来限制对 cache 的搜索，目

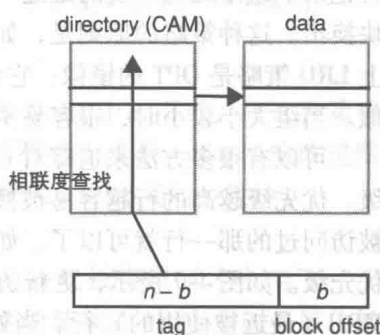


图 4-6 全相联 cache

录的标识是整个块地址。为了找到需要的块，要对所有目录项并行地进行比较。

读或写访问都需要两步。首先将块地址标识（即整个块地址）与全部目录项的标识进行比较。为了完成比较，目录被设计成基于内容寻址的存储器（Content-Addressable Memory, CAM）。RAM 与 CAM 之间的不同在于，RAM 是按地址访问的，而 CAM 按照其内容的某个域（在全相联 cache 中，这个域就是块地址）来访问。标识总线贯穿整个目录，线上的标识会并行地与每个目录的标识进行比较。这一比较操作由每个目录项中的比较器完成。

如果某个比较器检查出标识匹配，对应的数据存储器行就被激活，然后返回对应的数据。注意，不能像组相联 cache 那样将目录检查与取数据并行进行。此外，目录访问（CAM 访问包括信号在整个目录中的传播以及逻辑比较）要比 RAM 访问慢（需要地址译码）。由于全相联 cache 中的每个目录项都需要一个比较器，它的密度要小于 RAM。全相联 cache 的优点在于，由于内存块到 cache 的映射是灵活可变的，与组相联相比，其命中率会高一些。因此，在小一些的 cache 中全相联 cache 更受欢迎，因为在这类 cache 中热点组的竞争会影响系统的性能。全相联 cache 可以看作只有一个组的组相联 cache。

4.3.2 替换策略

当访问某个不在 cache 中的内存块时，该访问会触发 cache 失效并选择一个换出块进行替换。换出块所在的 cache 行必须是失效块的地址所映射到的 cache 行。在直接映射 cache 中，一个失效块只能被映射到唯一的 cache 行，因此换出块就很容易确定了。但是，在组相联 cache 和全相联 cache 中，会同时存在几个可以换出的块。在组相联 cache 中，任何一个处在映射组中的块都是备选的换出块。在全相联 cache 中，整个 cache 中所有的块都是备选的换出块。在组相联或全相联 cache 中选择换出块的过程称为替换策略。

最简单的替换策略是随机替换，在这种策略中，会随机选择组中的一个块作为换出块。随机替换决策的过程中不需要维护过去访问的信息，它与访问块的历史记录无关。

其他替换策略则试图选择一个换出块使得 cache 的失效率最小。如果可以知道未来对每个组的访问模式，那么最恰当的换出块应该是将来最久才会被访问的块。这种理想化的策略称为 OPT（最优）策略。OPT 策略的原则很简单，如果替换策略在一次失效时保留了某个块，那么它会希望将这块一直保留到下次对它的访问。如果无法将它一直保留到下次访问时，就应该换出它，因为它会毫无意义地占据 cache 空间。如果下次对某块的访问越迟，那么根据 OPT 策略，这个块越可能被换出。OPT 策略是无法实现的，因为它需要知道未来访问的模式。但是 OPT 策略可以给出失效率的下界，其他可行的策略可以以此为参照进行比较。在考虑到访问历史的可行策略中，希望过去的访问行为可以用来预测将来的访问行为。

最近最少使用（Least Recently Used, LRU）策略依赖于对每个块的历史访问的局部性，并且记录下组中对每个块的最近一次访问的时间。在一次失效时，它选择过去最久未被使用的那块换出。这种策略的原则是，如果某个块最近被访问过，那么将来也可能再访问它。某种程度上 LRU 策略是 OPT 的镜像：它希望根据局部性原理，过去访问的行为会与将来访问的行为类似。当组大小较小时，很容易实现 LRU 策略。

可以有很多方法来追踪对 cache 行的访问历史。方法之一是给每个 cache 行分配一个优先级，优先级越高的行越容易被换出。如果组大小是 2 行，那么只需要一个访问历史位指向最近被访问过的那一行就可以了。如果组大小是 4 行，那么每行需要两位来记录访问历史以及替换优先级。如图 4-7 所示，更新访问历史位的过程有点复杂。每行有两个历史位，优先级 0 对应 MRU（最近被使用的）行。当处在优先级 2 的第 3 行（Line 3）被命中时（图 4-7a），优先级低于 3 的历史位需要更新，而优先级为 3（Line 2）历史位保持不变。当出现失效时

(图 4-7b)，所有优先级位都需要进行模 4 加 1。当 MRU 行命中时，优先级位不会改变。由于历史位的更新比较复杂，人们经常使用伪 LRU 来替代 LRU 策略。伪 LRU 策略并不像 LRU 那样精确记录对块的访问历史，但是对历史位的更新会简单很多，尤其是对于较大的组。

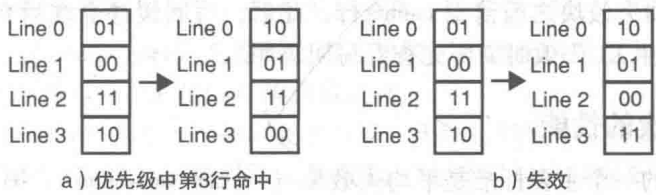


图 4-7 LRU 中更新访问历史位

LRU 和伪 LRU 策略的问题在于，每次访问 cache 时，无论命中还是失效都需要更新历史位。与此相反的是先进先出 (First In First Out, FIFO) 策略，它只会在失效时通过模 4 加 1 更新历史位 (此时有更多重叠的时间)。FIFO 策略的问题在于，即使有的块被频繁访问也会被换出。

4.3.3 写策略

在维护存储层次的一致性方面有很多不同的写策略，主要的策略包括写穿透 (write through) 和写回 (write back) 两种。

写穿透 cache

在写穿透 cache 中，所有的 store 操作都会更新到低级 cache。在流水执行的机制中，如果所有 store 操作的执行速度都按照低级 cache 的速度来执行，那么机器的 IPC 会受到很大的影响，因此，可以利用 store 缓冲来避免将低级 cache 的访问延迟暴露给处理器，如图 4-8a 所示。store 缓冲会挂起 store 操作，包括访问地址和数据，缓冲控制器会在通向下一级 cache 的总线空闲时传播地址和数据，只有当缓冲区满时才会暂停处理器。由于下一级的数据总是更新过的，当写失效时就没有必要重新给该块分配空间了。因此，写穿透 cache 在写失效时不会分配 cache 行，而是只将写操作插入 store 缓冲。为了维护一致性，load 失效时需要先查看 store 缓冲中是否有相同地址的 store 操作。如果检测到匹配，load 失效要么返回缓冲中的最新值 (前递 store 缓冲)，要么在访问下一级 cache 之前等待 store 缓冲中的操作执行完。

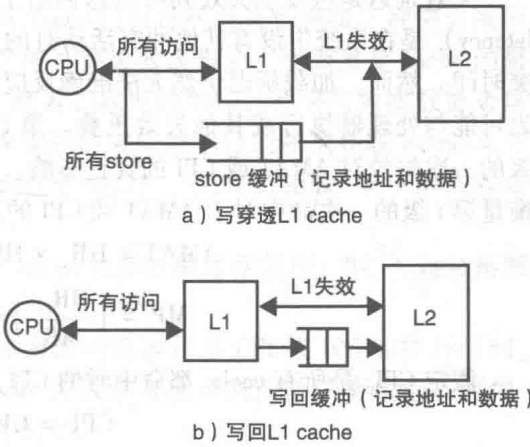


图 4-8 写穿透和写回 cache 的架构

写穿透策略可以简化 cache 设计，并且常被用在较小的一级 cache 中。不过，在这种策略下，即使 cache 增大，其所需的往下一级写的数据量也不会有所缓解，这是 cache 架构改进需要重点考虑的。对于大的 cache 来说，通常采用写回策略维护两级 cache 之间的一致性。

写回 cache

在写回 cache 中 (如图 4-8b 所示)，只有当被修改过的 cache 块被换出时才会向下一级 cache 传递写操作。每个 cache 行需要一个脏位，当失效后将一个块装载到 cache 中时，脏位被复位。当写操作修改 cache 行时，脏位被置位。当根据替换策略换出一个脏块时，必须将其写

回到下一级 cache 中, 而干净的块则不需要写回。由于 load 操作处在处理器执行的关键路径上, 应该尽快解决 load 失效。当发生 load 失效时, 先选取换出块并向下一级发出失效请求。当 load 操作被挂起时, 如果换出块被修改过, 换出块会被送到一个写回缓冲中 (也称作 victim 缓冲), 从而保证在返回失效块之前清空 cache 行。此后, 写回缓冲会在后台向下一级更新。为了维护一致性, 当发生 L1 失效时需要先查看写回缓冲。

4.3.4 cache 层次的性能

衡量 cache 性能的一个主要标准是平均失效率 (average miss rate)。第 i 级 cache 的失效率 (MR_i) 等于失效次数与处理器访问次数的比。在层次结构的 cache 中, 低于 L1 级的 cache 级的失效率也被定义为失效数与更高层次 cache 访问本级 cache 次数的比 (相对失效率), 第 i 级的这种失效率等于 MR_i/MR_{i-1} 。使用 MR 作为衡量标准的好处是, 它独立于高层次 cache 的失效率, 并且对 MR 的计算可以把当前 cache 当作 cache 层次中的唯一的 cache 计算。

第 i 级的命中率 (HR_i) 定义为处理器访问命中的比例, 等于 $1 - MR_i$ 。

失效率对于估计每次存储器访问 (指令或数据) 的平均时间很有帮助。但是, 它与 CPI 没有直接关系, 因为 load 和 store 指令 (需要两次存储器访问) 的比例是依赖于程序的。另一种常用的衡量标准是在第 i 级的每条指令失效次数 (MPI_i)。这个指标的计算方法是用总的失效数除以总的执行的 (提交的) 指令数。

失效率和每条指令失效次数在衡量 cache 性能上都是有所欠缺的, 因为它们没有考虑到失效对于性能的影响。一个可以囊括所有 cache 性能的指标是平均访存时间 (Average Memory Access Time, AMAT), 另一种指标是失效对于 CPI 的影响。

失效延迟是从发现失效到将值返回给上一层 cache 所用的周期数。未加载延迟 (unloaded latency) 是在系统中没有其他冲突活动时的失效延迟。加载延迟 (loaded latency) 则包括了冲突时间。然而, 加载延迟仍然无法准确反应一次失效对于处理器性能的真正影响, 因为失效延迟可能与处理器执行或其他失效重叠。第 i 级的失效开销 (Miss Penalty, MP_i) 可以衡量第 i 级的一次失效对 AMAT 或 CPI 的真正影响。同样, 第 i 级的命中开销 (Hit Penalty, HP_i) 可以衡量第 i 级的一次命中对于 AMAT 或 CPI 的真正影响。有:

$$AMAT = HR_i \times HP_i + MR_i \times MP_i \quad (4.1)$$

$$MP_i = \left(\frac{HR_{i+1}}{MR_i} \right) \times HP_{i+1} + \left(\frac{MR_{i+1}}{MR_i} \right) \times MP_{i+1} \quad (4.2)$$

假定 CPI_0 是所有 cache 都命中时的 CPI, 则

$$CPI = CPI_0 + MPI_i \times MP_i \quad (4.3)$$

在静态流水线 (如五级流水线) 中很容易计算出失效开销, 这种情况下, 处理器在 L1 失效时停顿一个周期, 并且一次处理一个 cache 失效, 可以通过每级的未加载命中和失效延迟将失效开销计算出来。但是, 在允许推测执行的处理器中就难计算出失效开销了。在乱序执行处理器中, 很难量化一次失效对于处理器执行的影响。当 load 发生失效时, 处理器并不会停顿, 它会尽可能执行一些不相关的指令。由于相关的指令无法执行, 失效可能在某种程度上降低处理器速度 (甚至降到很低), 但这种降低是很难估计的。任何给定访存指令的开销都依赖于处理器中其他同时执行的指令的活动。因此, 通常来讲, cache 的改进可能并不会直接导致 CPI (或执行时间) 的改善。例如, 对 cache 失效率的简单改进甚至可能会导致 CPI 变差, 即使其他所有参数都保持不变, 这是因为乱序执行是无法预测的。因此式 (4.1) ~ 式 (4.3) 在现代系统中的应用十分有限。

4.3.5 cache 失效的分类

并不是所有 cache 失效都是一样的,相反,cache 失效有不同的种类,并且有不同的措施(软件的或硬件的)来降低这些失效。一种常见的分类方法是 3C 失效:

- 冷失效(cold 或 compulsory)是指每个内存块第一次访问时发生的 cache 失效。冷失效是无法避免的,除非 cache 将内存块预取进来。
- 容量失效(capacity)是由于 cache 大小不够,无法容纳整个工作集的数据而引起的。
- 冲突失效(conflict)是由于映射策略的限制,使得多个块映射到组相联 cache 中的相同的组而产生的失效。

在给定工作量和硬件条件的情况下,经常使用 cache 模拟来计算每种失效的次数。通常情况下,简单的 trace 驱动模拟就足够了。

为了确定冷失效的次数,通过将工作集在拥有无限容量 cache 的目标机上进行模拟。冷失效的次数与大多数 cache 参数无关,如 cache 大小、组织方式以及替换策略。可以通过增加 cache 块大小的方式来减少冷失效次数,因为随着块大小的增加,一次失效可以取更多的数据。通常情况下冷失效率在总的失效率中可以忽略不计。

为了确定容量失效的次数,通过将工作集在使用全相联 cache 的目标机上进行模拟。容量失效的次数等于总次数减去冷失效次数。影响容量失效次数的主要因素是 cache 大小。然而,其他因素(如块大小)也很重要。随着块大小的增加,冷失效率得以改进,因为可以开发更多的空间局部性。但是,在有限的容量下,存在某个临界点,使得空间局部性带来的好处可以被对同一 cache 行的竞争所抵消掉。计算容量失效的一个重要因素是替换策略,显然最优替换策略(OPT)是首选,因为它可以使全相联 cache 的失效次数最小。

为了确定冲突失效的次数,通过将工作集在实际 cache 的目标机上进行模拟。冲突失效的次数等于总次数减去冷失效次数和容量失效次数。影响冲突失效次数的主要因素是 cache 的组织方式。直接映射的 cache 拥有最高的冲突失效率。随着相联度的增加,冲突失效次数显著降低。有数据表明八路组相联的 cache 对于大多数工作集的冲突失效率已经和全相联 cache 几乎相同了。为了清晰描述冲突失效的重要性,有个经验性法则指出,两路组相联 cache 的失效率同容量二倍于它的直接映射 cache 差不多。

3C 失效只是强调单处理器中的失效率。在含有 cache 的多处理器系统中,3C 失效会拓展到 4C 失效,其中第四个 C 指一致性失效。

尽管失效率是 cache 性能的一个重要指标,但是最终的标准仍是 CPI 或 IPC 和执行时间。接近于执行时间的指标是 AMAT,它包括了改进 cache 失效率方法对时间的所有影响。例如,更大的块会增加失效开销,更高的相联度会增加命中开销。

设计 cache 层次是为了使命中开销、失效率和失效开销最小化。为了最小化失效开销,可以通过更快的电路、更高的并发性和流水化存储系统来减小延迟。此外,也可以通过将多个 cache 命中和失效重叠,在访问之前预取块到 cache,来降低失效延迟。

4.3.6 非阻塞 cache

对于静态流水线(如五级流水线)而言,cache 一次只处理一个处理器的访问是可以接受的,但是对于可以在一个周期执行多条访问指令的机器来说,这就会阻碍速度的加快。同一时刻只接受一个处理器请求的 cache 称作阻塞式 cache(blocking cache)。现代的 cache 都是非阻塞的(non-blocking,或者叫 lockup-free),它们可以在同一时间处理多个命中和失效。为了理解为什么非阻塞是可行的,需要注意到 cache 是一个需要两侧交互的设备。cache 的功能可以被

分到两个控制器中：一个控制到来的请求，一个处理 cache 失效。图 4-9 给出了非阻塞的 L1 cache，它需要响应来自处理器的请求，同时需要处理到低级 cache 的失效。

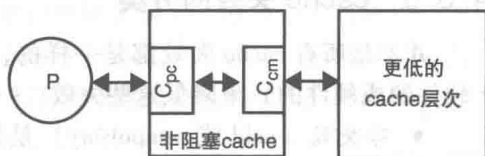


图 4-9 非阻塞 L1 cache: C_{cm} 为 cache 到内存的接口; C_{pc} 为处理器到 cache 的接口

cache 在每一侧都有一个控制器，命中的请求可以由一个控制器完成，如果请求未命中，它就被交给第二个控制器来处理失效。每个核的 L1 和 L2 cache 都需要这样，此外，多线程共享的 cache，如支持多线程核的 L1 cache 和片上多处理器的共享 L2 cache，也需要这样做。一旦控制器处理一个新的失

效，就会分配一个 cache 行，换出一块，换出块可能先存入写回缓冲中等待写回。控制器必须记录下所有未完成的失效，这一功能由失效状态处理寄存器 (Miss Status Handling Register, MSHR) 完成。每一个未完成的失效都会分配一个 MSHR。MSHR 中包含诸如块地址、对应的 cache 行以及 load 指令的目的寄存器等信息，这些信息可以用来完成失效处理，以及避免对同一块发送多个失效请求。

相比于阻塞式 cache，非阻塞 cache 会导致更多的失效，因为尽管对某个块的失效已经在处理中，非阻塞 cache 仍会接受对这个块的访问，很多访问失效同那些未完成的失效所对应的块是一样的。第一次失效叫作初次失效，并且已经分配了 MSHR，失效请求已经发送给低级别的存储层次。在现代处理器中，主频达到了几 GHz 的水平，返回一个访问失效的块需要 20~200 个周期。在这段时间内，由于访存的局部性，可能会有更多对该块的访问。当非阻塞 cache 收到一个对未完成的失效块的访问时，它通过分配一个 MSHR 记录下这次访问，但不会向下一级存储发送请求，因为这一块的初次失效已经在处理中了。对已经在处理中的块的失效访问称作二次失效。当一个失效块返回时，初次失效和二次失效请求的字都由 cache 提供，这样二次失效的延迟有一部分与初次失效重叠了。二次失效的情况在阻塞式 cache 中不会出现，因为阻塞式 cache 一次只接收和处理一个访问请求。

非阻塞 cache 在现代的宽发射乱序执行处理器中很常见，因为必须维护内存操作的吞吐量，并且内存访问必须与失效和处理器执行并发执行，从而隐藏 cache 失效的开销。此外，通过预加载和预取 cache 块来避免失效也需要非阻塞 cache。

例 4.1 阻塞 cache 和非阻塞 cache 的比较 为了比较阻塞 cache 和非阻塞 cache 的吞吐量，考虑下面的微型基准测试程序：

```

TOY:  LW R1,0(R2)
      ADDI R2,R2,#4
      BNE R2,R4,TOY
  
```

假设 cache 容量无限，块大小为 16 字节 (4 个字)，初始时空。处理器速度很快，以至于 load/store 队列总是满的，地址总是可用的，从而每个周期都有一个 load 指令可以发射到 cache。

首先考虑阻塞 cache，一次命中需要 1 个周期，一次失效需要 200 个周期 (包括最初检查 cache 的访问)，执行过程如图 4-10a 所示。每个失效需要 200 个周期，并且后边紧接着有 3 次命中。循环的每次执行大概需要 50 个周期。现在假设 cache 是非阻塞的，并且有 16 个 MSHR，因此它最多可以运行 16 个未完成的失效 (初次失效和二次失效)。其执行过程如图 4-10b 所示。P 和 S 指向初次失效或二次失效发生和返回的周期数。相同的模式每 200 个周期循环一次，每 200 个周期执行 16 次 load，因此一次迭代的执行大概需要 13 个周期。

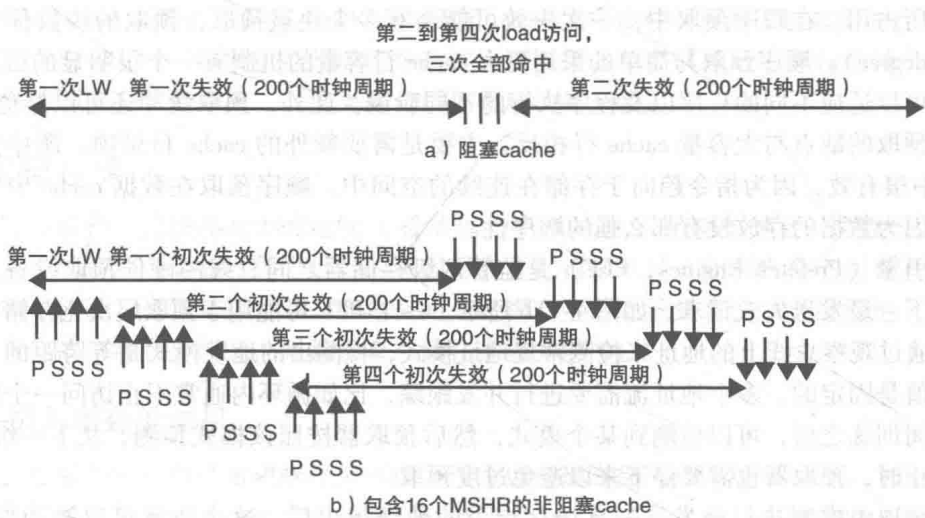


图 4-10 两种 cache 的吞吐量比较

4.3.7 cache 预取和预加载

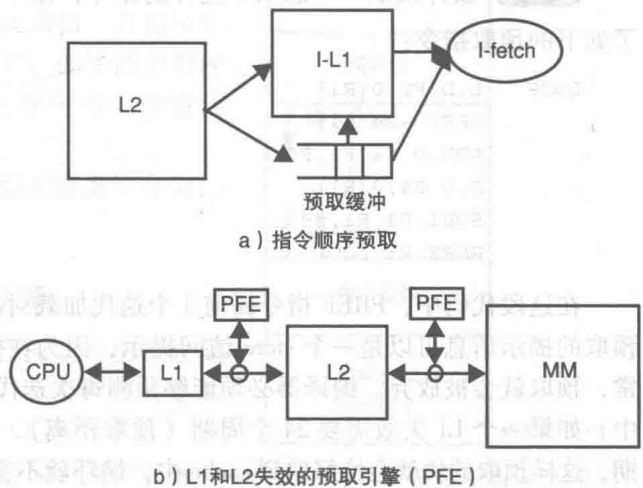
在乱序推测执行处理器中支持非阻塞 cache 可以保证获得很高的内存吞吐率。但是，每次内存访问时的初次失效及其开销还是无法避免。如果在访问某一块之前先将其预取到 cache 中，就可以避免初次失效以及之后对于同一块的二次失效，这就是预取和预加载的基本思想。“预加载”（preloading）和“预取”（prefetching）意思基本相同，但“预加载”主要用在第一次向 cache 中加载块的情况（冷失效）。

cache 预取有多种不同的形式，可以通过软件或硬件完成，指令和数据都可以被预取。所有的预取都是无约束力的（non-binding），且只用在 cache 中。这表明预取只会简单地将被预取块以及正确的访问权限返回到 cache（或者与 cache 相关联的缓冲）中，而不真正执行访问操作。返回异常的预取操作会被放弃。下面先讲述一些常用的硬件预取技术。

硬件预取

预取指令是很重要的，因为 I-cache 失效会导致处理器没有指令可执行。常用的预取机制是顺序预取，在顺序预取中，会预取当前访问失效块的相邻块（在地址空间中相邻）。I-cache 有一个小的缓冲区用来保存几条指令（通常是一个块大小），如图 4-11a 所示。当 cache 失效时，除了取回失效的块之外，还会检查 I-cache 中是否有相邻的下一块（相邻地址空间），如果该块不在 cache 中，就将其预取到预取缓冲中。如果这个块在预取缓冲中，就将其加载到 cache 中，并释放预取缓冲中的空间。

将块预取到小的预取缓冲而不是直接取到 cache 中，可以避免无用的预取，比如取到不被访问的块。如果预取缓冲中的块不被访问，它所占的空间会被其



b) L1和L2失效的预取引擎（PFE）

图 4-11 指令和数据的硬件预取

他预取块所占用。在顺序预取中，一次失效可能会有多个块被预取，预取的块数称为预取度 (prefetch degree)。顺序预取与简单的采用更大 cache 行容量的机制有一个很明显的区别，预取在块级别可以适应不同的程序以及程序执行的不同阶段。此外，预取缓冲还可以避免 cache 污染。顺序预取的缺点与大容量 cache 行相反，主要是需要额外的 cache 目录项。顺序预取在指令 cache 中很有效，因为指令趋向于存储在连续的空间中。顺序预取在数据 cache 中就没那么有效了，因为数据的存放没有那么强的顺序性。

预取引擎 (Prefetch Engines, PFE) 是监管两级存储器之间总线的硬件预取设备，同时它也会自动向下一级发送失效请求，如图 4-11b 所示。一个 PFE 可能用于预取层次化存储中的每一级。PFE 通过观察总线上的地址来检测常规地址模式，检测出的地址模式是等跨距的，即地址之间的差值是固定的。多个地址流需要进行并发跟踪，比如循环内通常不止访问一个数组。经过一段时间训练之后，可以检测到某个模式，然后预取器按照该模式预测，从下一级预取块。当模式停止时，预取器也需要停下来以避免过度预取。

硬件预取由推测执行触发，一旦访存指令的地址得出后，这个块就可以被预取到 cache 中，即使该行为会破坏程序的正确执行也没关系。例如，store 指令可以在得到地址之后就进行预取，尽管它们本应该在到达重排序队列顶端才被执行。load 指令在得到地址之后也可以预取，即使之前的 store 指令地址可能还不知道。之所以能这么做，是因为预取只发生在 cache 内并且是非约束性的。一旦某个失效触碰到这个块，它就会被预取到 cache 中。如果预取时发生异常，这次预取就被丢弃掉。通过对访问的预测，cache 块很可能在真实访存指令执行之前或者在重排序队列提交之前就被预取到 cache 中。这种硬件预取是无害的，只是在预测错误必须撤回的情况下可能会污染 cache。

软件预取

cache 预取也可以通过软件触发。程序员和编译器由于懂得代码含义，它们可以知晓未来执行的情况。因此编译器 (可能需要程序员的帮助) 可能显式地插入预取指令到代码中。预取指令类似于 load，但是它们没有明确的目的寄存器。预取指令必须是非阻塞的指令，例如，在五级流水线中，预取指令如果引起 cache 失效也不会导致流水线停顿。预取指令的形式有很多种，每种都包括一些提示信息，告诉硬件在什么状态下预取哪些东西。例如，有对 load、store 或流指令的预取指令，指令和数据都可以预取。为了使软件预取有效，cache 必须是非阻塞的。

例 4.2 软件预取 考虑如下这样的循环，循环中给向量中的每个双字加一个常数，插入了如下的预取指令：

```

LOOP   L.D F2,0(R1)
        PREF -24(R1)
        ADD.D F4,F2,F0
        S.D F4,0(R1)
        SUBI R1,R1,#8
        BNEZ R1,LOOP
    
```

在这段代码中，PREF 指令提前 3 个迭代加载对应的 cache 块。预取的操作数是内存地址，预取的提示信息可以是一个 store 访问提示，因为在循环体中数据块进行了更新。如果出现异常，预取就会被放弃。编译器必须能够预测每次迭代的周期数，从而决定预取的布局。在本例中，如果一个 L1 失效需要 24 个周期 (预取距离)，并且除去失效后执行一次循环需要 6 个周期，这样预取的块就会恰好取到 cache 中，循环就不会失效。对覆盖预取距离的迭代执行次数进行估计，通常是很难的。如果低估了迭代次数，load 指令就需要等待失效开销，如果高估了迭代

次数，可能预取到的块会被替换掉，这样既污染了 cache 又可能使处理器等待整个失效开销。

每次迭代取值、译码以及执行预取指令这些固定开销必须在其他地方进行均摊。理想情况下，有条件的预取会带来好处，但这无疑会增加循环的指令数。

4.4 虚拟存储

在虚存系统中，处理器发出的地址（通常称为有效地址）是虚拟地址，需要先转换为物理地址才能访问物理存储器。每个运行的进程都会分配虚拟地址。物理内存空间是真正由存储介质组成的存储区，通常是指主存。虚存空间是进程产生的地址集合，一个进程绝大多数的虚拟空间都是空的、没有分配的，并且没有存储在物理介质中。

4.4.1 引入虚存的动机

设计虚存最初的动机是希望程序员可以不用考虑物理存储大小来编写程序，尤其是程序员编写的代码长度和数据区大小可以超过物理存储的大小。在虚存之前，程序员需要在代码中显示地管理物理地址空间，这一技术也叫作程序叠加（overlay）。程序员通过编写 I/O 操作将大块代码和数据装入内存，当执行完当前的指令块后，后面的指令和数据块就会叠加覆盖当前的存储区域。很显然，这是一个非常繁琐、容易出错并且很难调试的方法。此外，针对不同存储大小的机器需要编写不同的程序，因此可移植性也很差。

自动管理程序存储空间有很多好处，下面列举一些这样做的好处。

- 代码和数据在主存中可以重定位。代码和数据在虚存空间中分配地址，并且不随物理地址的变动而改变。
- 一个程序分配的地址独立于物理地址和主存大小，这样可以灵活且有效地让多个程序共享一个系统。这种方法在通过软件多线程来隐藏 I/O 延迟中很重要。
- 每个程序的地址必须通过它自己的转换表进行转换，通过保证每个程序只访问自己的地址空间，就可以使程序之间相互隔离。这是限制用户权限的关键。
- 沿着从虚拟地址转换到物理地址的路径，内核可以采取额外的控制，这是检查程序是否执行混乱的关键，如将数据当作代码执行或将代码作为数据访问等错误，也是防止用户受自身错误影响的关键。
- 最后，虚存有利于在物理内存中的多个进程之间共享数据和代码，这样做可以支持进程间通信，并且可以通过共享提高物理内存利用率。比如，如果两个程序代码相同，执行时就没有必要在主存中拷贝两份代码了。

上述这些优点也是今天绝大多数机器都支持虚拟存储的原因。

4.4.2 从操作系统视角看到的虚拟存储

在现代操作系统中，每个进程都有自己独立的虚拟存储空间，操作系统内核根据进程上下文执行。每个程序的虚拟空间分为两大部分：内核（或系统）区以及用户区，如图 4-12 所示。内核区和用户区的分界是固定的，用户区通常分为 3 个段：代码段、数据段和堆栈段。对内核区的虚拟地



图 4-12 进程虚拟地址空间布（32 位地址空间）

址映射，所有程序都一样，但是对用户页面的映射则不同。因此，通常情况下，每个程序都有一组私有的用户区地址转换，还有一个全局的内核区地址转换。

每个进程的虚拟空间是由编译器静态和操作系统动态共同分配的。编译器在编译程序代码并且分配地址时，分配代码段和静态数据段。诸如堆栈的动态空间是通过缺页机制动态分配的。随着栈或堆的增长，如果到达了之前未分配的页，就会引起缺页，进而分配一个新页。虚存可以通过 `malloc` 语句动态分配虚拟内存。整体上每个进程的虚地址空间的代码和数据都很稀疏，尤其是对于 64 位的处理器，存在很大的空白。

虚拟内存按页进行组织，物理内存按照与页大小相同的页帧进行组织。页和页帧可以类比于内存块和 cache 行，因此，主存相当于硬盘上存储的虚存页的 cache。页对页帧按需分时复用（通过页面需求算法），所以当执行进程需要时，会将页设备（通常是硬盘）上的页换到主存。对不在内存中的页面的访问会触发一次缺页，处理器按照缺页异常进行处理。当缺页异常触发时，处理器会直接执行软件处理程序，这个程序称为缺页处理程序。这个程序首先会在主存中找到一个替换页，如果这个替换页被修改过，它会被换出（也就是说写回硬盘），然后将新页换入到释放的页帧。因此维护一致性的策略是写回。

主存相对于虚存来说是一个全相联的 cache，这是因为虚拟页可以放到任何一个页帧中。考虑到整个物理内存的大小，替换策略不能采用 LRU。FIFO 策略是相对更容易实现的策略：内核维护所有页帧的一个 FIFO 队列，这个队列按照填充新页的顺序，每次选取队列顶端的页帧进行替换。一种常用的替换算法是工作集或者其变种。工作集记录了在最近的时间窗口内访问的页帧轨迹，在时间窗口内未访问的页面作为换出的候选页，将其标记为“无效”并插入到页 cache 中。当内核需要一个新的页帧时，它就从页 cache 中选取一个。在页 cache 中的页面仍驻留在主存中，并且可以在缺页的情况下快速重新访问。因此这里有硬件缺页和软件缺页的区分：发生硬件缺页时，页面必须从硬盘上取，并分配一个新的页帧；而发生软件缺页时，页面已经在页 cache 中，缺页处理程序只需将相应的页表项置位有效即可。

图 4-13 描述了虚实地址空间的映射。两个 32 位的虚地址空间映射到物理内存中。物理内存空间可能比每个虚地址空间都大，但是，由于很多程序经常同时运行，总的虚地址空间会比物理地址空间大很多。总的来说，在不同虚地址空间，两个一样的虚地址映射到不同的物理地址。处理器 1 和 2 的地址 VA3 被称为同名的（名字相同，但指向对象不同）。同名在存储系统中会造成混乱，一个简单的消除同名的方法是在虚地址前加上进程 ID（PID）来保证它们唯一性。地址 VA1 和 VA2 称为别名（名字不同，但指向相同的对象），为了灵活地共享代码和数据，需要支持别名。别名也会造成存储系统的混乱，并且比同名更难解决。跟别名相关的问题称为别名问题，我们会在后面的部分进行讲解。

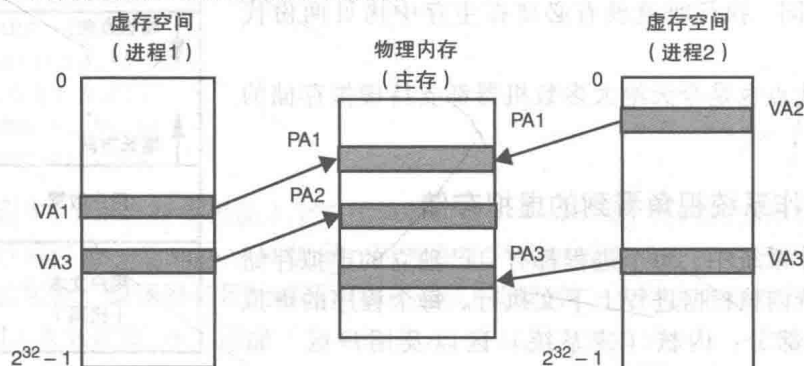


图 4-13 虚实内存地址映射

4.4.3 虚地址转换

支持虚存的主要结构就是硬件的地址转换，需要动态且有效地将每个虚地址转换为一个实地址。存储地址转换的表存放在内存中，称为页表（page table）。

图 4-14 给出了地址转换过程，图中用一个页表就完成了所有地址转换，这是一个简化了的情况。页表存在主存中（这部分由物理地址访问）。每个页表项包含物理页号（或页帧号）以及状态位和控制位，因此一个页表项可能是一个 32 位的字。每个活动进程都有一个页表，有一个寄存器保存了页表基址（PTAB），这个地址也是程序状态的一部分。虚地址（VA）分为两部分：虚页号（VPN）和页内偏移。页内偏移并不作为地址转换的一部分，因为这部分的虚实地址是相同的。为了将虚页号转换为物理页号，需要将虚页号（乘 4）加上 PTBA 的内容去访问主存，取得页表项（PTE），然后将物理页号与页内偏移连接起来就得到了物理地址（PA）。

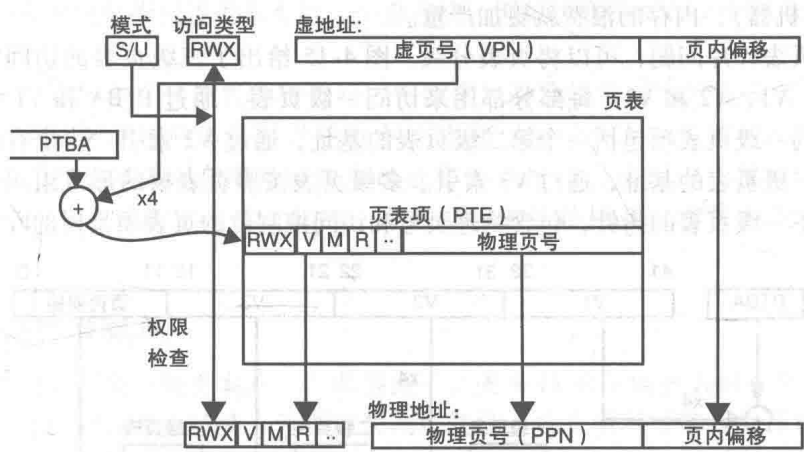


图 4-14 通过页表进行地址转换

4.4.4 访存控制

每次访问指令或数据都必须进行地址转换，因此需要在页表项上增加状态位来控制访问。考虑到每个页表项都是进程私有的，这些状态位只能由本进程进行设置。

读/写/执行（RWX）位用来进行权限检查，它们标记该页的内容是只读数据、可读可写数据还是可执行代码。处理器有两种状态：特权态和用户态，S/U 位用来记录对应状态。当用户进程陷入内核态时进行置位，当继续用户进程执行时进行复位。在特权态（内核态），处理器可以修改所有资源，包括页表。在用户态，访问虚存中的页受限，并且由页表中的访问权限位控制。保护硬件会检查处理器的 S/U 状态以及处理器提交的请求类型（读、写或执行），在用户态时需要将请求类型与 RWX 域比较。如果处理器企图修改只读数据，将代码当作数据读写或将数据当作代码执行，处理器就会陷入特权态。

页表项的一些状态位被用来进行虚存管理：

- V 位（有效位）表明该页是否在主存中。如果 V 位置位，则物理页号域内就是该页的物理页号，如果复位，则该页就不在主存中甚至从未被使用过，会触发缺页异常。
- D 位（脏位）表明该页调入主存后是否被修改过。当页被换进时该位复位，第一次写该页时置位，它可以用来减少换出的数据量。
- R 位（引用位）被内核用来实现替换算法。它可以被内核复位，并且每次访问时都会置位。

可以增加其他状态位来控制 cache 内的访问, 因为页表项的访问是所有访存操作在访问 cache 之前的必经步骤。比如, 一些页可能是不可缓存的, 这种情况下, 对它们的访问可以绕过 cache。而不同的可缓存页也可能有不同的写策略。在多处理器系统中, 页可以是一致的也可以是不一致的, 或者对不同的页可以有不同的缓存协议。

4.4.5 多级页表

通常进程都不会用完整个虚拟地址空间, 相反, 如图 4-12 所示, 每个进程的虚地址空间一般都是非常稀疏的。由于页表按照图 4-14 的方式访问, 即将虚页号加上一个基址, 因此所有的页表项都需要存在主存中, 即使大部分页表项是无效的, 这就是所谓的页表碎片问题。例如, 一个 4GB 的虚地址空间包括 2^{20} 个 4KB 的页, 如果页表项大小为 4B, 每个程序就需要 4MB 的空间来存放页表, 考虑到有多个程序在同时执行, 这个开销是很大的。随着虚页号位数的增加 (比如 64 位机器), 内存的浪费就更加严重。

为了解决页表碎片问题, 可以将页表分级。图 4-15 给出了三级页表的访问方法, 虚页号被分为 3 部分, V1, V2 和 V3, 每部分都用来访问一级页表。通过 PTBA 和 V1 访问第一级页表。一个有效的一级页表项包括一个第二级页表的基址, 通过 V2 索引。一个有效的二级页表项包括一个第三级页表的基址, 通过 V3 索引。多级页表按照页表树的形式组织, 每个中间页表都包含指向下一级页表的指针, 包含物理页号和访问控制位的页表项是树的叶节点。

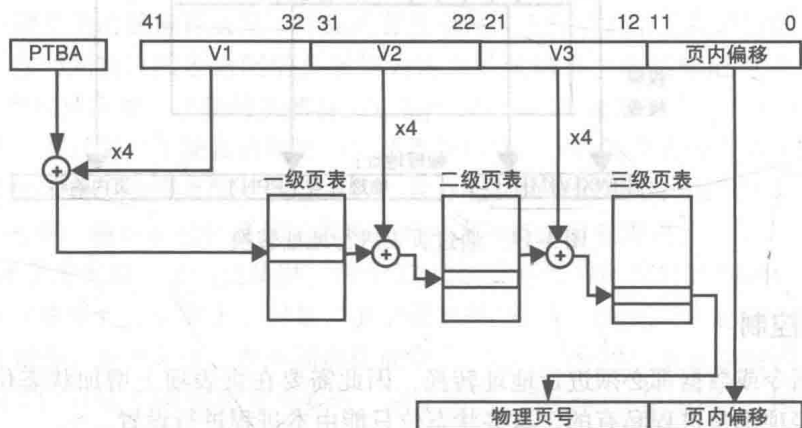


图 4-15 多级页表

不同级的页表项覆盖了不同数量的连续虚存空间。同样还假设页大小为 4KB, VPN 每部分的长度为 10 位, 因此虚地址长度为 42 位。一个一级页表项覆盖了 4GB 内存, 一个二级页表项覆盖 4MB 内存。由于虚存空间十分稀疏, 只有少数表项分配在二级和三级页表。因此, 多级页表需要的内存总量是几十 KB。如果用了一整个页表, 如图 4-14, 就需要 4GB 来存放页表。

这种大幅的空间节省主要得益于很多页表项都是空的, 为了节省空间就需要在每次地址转换时多访问几次页表。如果页表项都是有效的, 那么多级页表会比单一页表效果更差。因为随着页表树的层数增加, 定位次数和访问页表项的开销也会随之增加。

多级页表的结构有助于支持超级页 (superpage), 超级页由连续的多个页面组成。当数据或代码段会覆盖很多页时, 分配一个超级页给它们可能会有好处。如果内存中有足够的连续空间, 一个超级页在一次缺页中就可以加载到内存。在多级页表中, 包含物理地址和状态位的页表项不一定再是一个叶节点, 任何一层页表都可能包含物理地址和状态位。其中有一位表明该页表项是指向另一个页表的基址还是包含了转换地址。例如, 图 4-15 给出的页表, 可以支持

两种超级页：4MB（一个二级页表项）和4GB（一个一级页表项）。当然，内存中必须有足够的连续物理页帧来存储整个超级页。

4.4.6 反向页表

在多级页表的组织中，每个进程有自己的一组页表，其中仍然有很多页表项是无效的。一个彻底的解决方法是改变页表的结构，将它们反转过来。不再对应每个虚页有一个页表项，反向页表对应每个物理页帧有一个页表项，因此它是所有进程共享的，页表的大小与物理页帧数成正比，而不再与虚拟页成正比。这样做的动机在于页表项的数目不会超过页帧数。为了访问反向页表，虚页号通过哈希的方法指向共享页表中的某个表项。不同的反向页表于对冲突的处理方法不同，冲突是指多个虚页号有同样的哈希值。因为页表是共享的，所有进程用相同的虚地址访问相同的页，因此消除了同名问题。为了保证不同进程的访问权限，内存系统通常分成多个段，每个有效地址指向进程私有的一个虚存段，但是所有段都在一个虚存空间中，由所有进程共享。

在 PowerPC 架构中，共享表的项数比页帧数多。哈希函数得出的值指向了一组固定的连续页表项，有相同哈希值的虚页号共享这些页表项。这些页表项中的每一个都必须包括页帧号和虚页号，并且需要逐个搜索比较虚地址是否匹配。当出现一个缺页，并且对应的页表项已经满时，就必须换出一项。此时，换出页的地址转换信息将从页表中删除，但该页仍在主存中。

4.4.7 旁路转换缓冲

多级页表是时间和空间折中的一个典型例子。图 4-15 所示的页表组织形式，每次都需要访问 3 次内存才能完成地址转换。为了加快地址转换，页表项可以像数据那样放入 cache。由于每个页表项覆盖了很大一块内存，通常缓存的页表项命中率很高。不过，即使转换时每次都命中，仍需要访问几次 cache 才能完成一次转换。因此，每次对指令或数据的内存访问都至少需要 4 次 cache 访问，这是难以接受的。

解决这个问题的办法是用一块专门的存储记录下最近进行的地址转换，这块存储称为旁路转换缓冲（Translation Lookaside Buffer, TLB），或简称为转换缓冲（Translation Buffer, TB）。TLB 相当于页表项的 cache，通过虚页号进行访问，如果命中则返回页表项。有效的 TLB 项指向在主存中且有效的页。TLB 像 cache 那样组织，可以是直接映射、组相联或全相联的，如图 4-4、图 4-5 和图 4-6 所示，只是它们需要通过虚页号访问。TLB 的每一项包含一个标识（通常是虚页号的最高几位）以及一个有效位。为了避免同名问题，需要为标记加上扩展的 PID 号。对应的内存数据包括带有页帧号和状态位的页表项。图 4-16a 给出了组相联 TLB 的表项。“Misc”域包括一些额外的状态位用来控制 cache 访问。TLB 也可能是全相联的。需注意的是，TLB 比 cache 小很多，命中率高很多，这是因为一个 TLB 项覆盖了很大一块内存。

如图 4-16b 所示，在一次访存中，虚页号的最低几位用来索引 TLB 组。PID/VPN 标识域需要同当前 PID 和处理器的虚页号进行比较，此外还需要检查 V 位。如果命中，就将物理页号发送到 cache，并检测 TLB 项的状态位。如果 TLB 失效，在完成访问之前，必须将页表重新加载到 TLB 中。TLB 失效可以通过陷入异常并执行 TLB 失效处理程序来完成。CPU 上执行的处理程序会在主存中查找对应页表，并将对应的转换关系存到 TLB 中，然后进程恢复执行，再次访问时就会在 TLB 中命中。当 TLB 失效处理程序在内存中发现缺页时，就会跳转到缺页处理程序执行。这种方法很灵活，因为页表的变化可以很容易地通过修改软件处理程序来匹配。不过，在现在的乱序执行处理器中，异常处理的性能开销非常高。

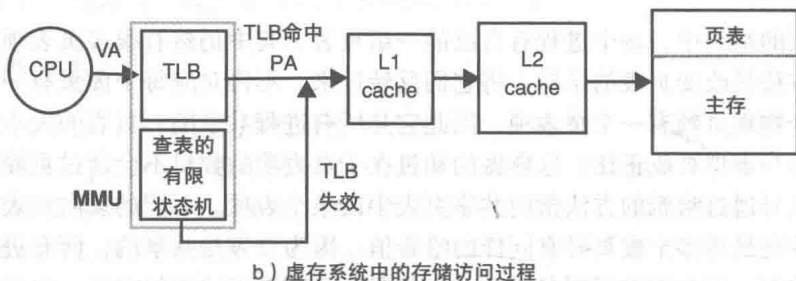


图 4-16 采用 TLB 的存储层次

如果能够采用特殊的微代码和硬件辅助 TLB 从内存中的页表直接加载页表项或更改状态位, 那么 TLB 失效就可以做到对处理器完全透明。这种情况下, TLB 需要一个查表的有限状态机 (Finite State Machine, FSM), 如图 4-16b 所示。TLB/FSM 的结合通常被称为是一个内存管理单元。如果 TLB 失效, 内存管理单元 (Memory Management Unit, MMU) 的微代码就会从页表中加载失效的表项, 然后访问继续进行。处理器不会察觉到发生了 TLB miss, 只是内存访问的时间由于查表而延长了。如果 MMU 检测到发生了缺页, 处理器就会陷入异常处理。

从逻辑上讲, TLB 必须位于处理器和 cache 之间, 这样做带来的结果是每次访问数据 (或指令) 都至少需要一个额外的周期, 这是我们不希望看到的。为了隐藏 TLB 访问延迟, TLB 访问和 cache 访问可能流水进行。图 4-17 给出了在五级流水中增加指令和数据 TLB 访问流水级的示意图。增加的流水级隐藏了在 cache 之前访问 TLB 的开销, 但是延长了 load 的执行时间和流水线, 当碰到预测错误的分支指令时, 会导致开销更大。

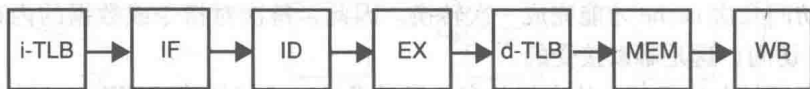


图 4-17 增加了流水访问 TLB 和 cache 的五级流水

另一种方法是并行访问 TLB 和一级 cache, 如图 4-18 所示。并行访问 TLB 和 cache 是可行的, 因为访问 cache 有两个阶段 (cache 索引和标识比较)。假设 L1 cache 的组索引位在页内偏移中, 那么这对物理地址和虚地址都是相同的, 也就是说这部分都是物理位。因此 cache 访问的第一阶段可以与 TLB 转换并行。在第一阶段结束后, 来自 TLB 的物理地址和物理 cache 标识都已经得到, 可以进行比较了 (假设 TLB 访问时间不比 cache 索引时间长)。注意 L2 cache 通常是用物理地址访问的, 这是因为访问 L2 时, 虚实地址转换早已经完成了。

并行访问 TLB 和 L1 cache 隐藏了 TLB 命中的延迟。但是, L1 cache 的大小就会受页内偏移 (即页大小) 所限制。通过对地址索引位的简单计算可以发现, cache 的相联度限制了 cache 的大小。比如, 直接映射 (一路组相联) cache 的最大容量只能为 1 页, 同样, 16 路组相联 cache 的最大容量为 16 页。

4.4.8 带物理标识的虚地址 cache

带物理标识的虚地址 cache 是物理地址 cache 的简单扩展, 如图 4-16b 和图 4-18 所示。不同的是, 这里的 cache 是通过虚页号的某几位来索引的。但是, cache 的标识 (tag) 仍是物理地址, 二级 cache 也还是通过物理地址访问。

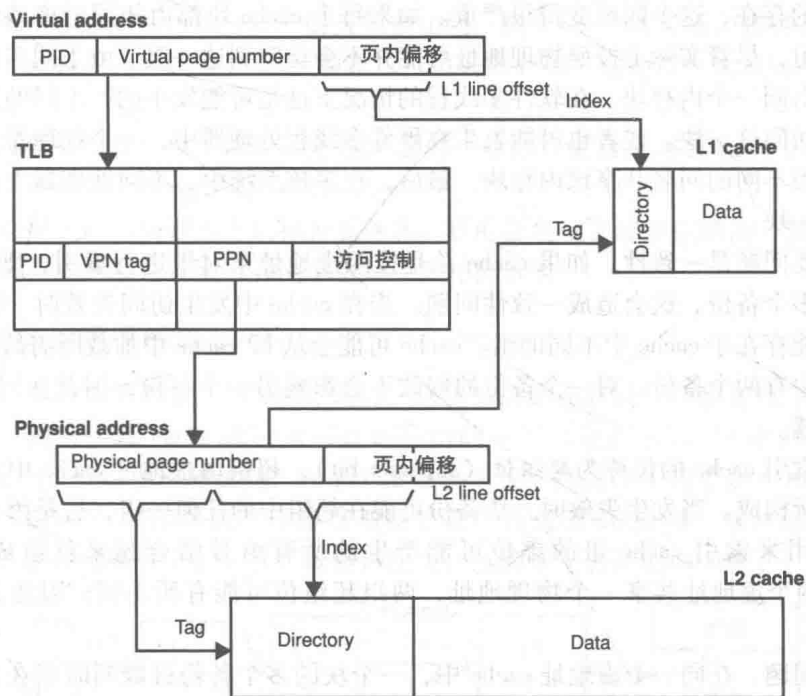


图 4-18 并行访问 TLB 和 cache

如果 L1 cache 大小超过上面的限制，则组号中最高几位用来索引 L1 cache 的就是虚位，如图 4-19a 所示。但是，对应的物理位在 TLB 转换完成前无法得到，因为这几位在虚实地址中的值没有统一的联系。

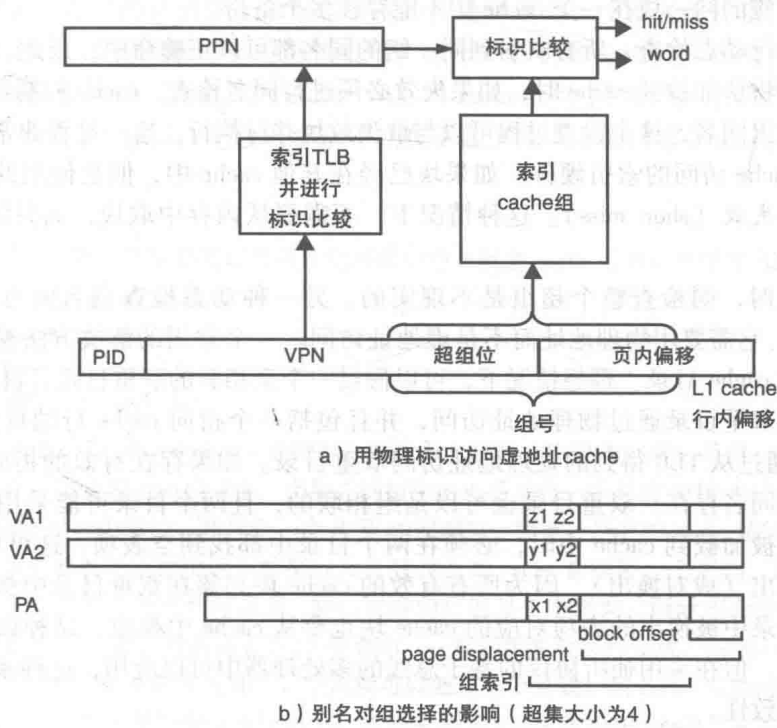


图 4-19 带物理标识的虚地址 cache

由于同名的存在,这个问题变得很严重。如果每个 cache 块都由相同的虚地址访问,则总会索引到同一组,尽管实际上按照物理地址可能并不会访问到这一组。由于同名问题,不同的虚地址可能索引同一个内存块。在软件多线程的情况下这是可能发生的:不同的进程轮流使用不同的虚地址访问这一块。或者也可能发生在硬件多线程处理器中,一个处理器上多个线程并发地执行,利用不同的同名共享该内存块。最后,在多核系统中,不同处理器上的进程也可能用别名访问同一块。

这里的主要问题是—致性。如果 cache 总是按照虚地址来对组进行索引,则 cache 中可能会有内存块的多个备份,这会造成一致性问题。当在 cache 中发生访问失效时,同一块由于不同的虚地址可能存在于 cache 中不同的组。cache 可能会从 L2 cache 中加载所需的块,导致同一块在不同的组中有两个备份。对一个备份的修改不会影响另一个备份,因此在同一 cache 中出现了—致性问题。

虚地址中索引 cache 的位称为超组位 (superset bit),超组由虚地址 cache 中块备份可能映射到的所有组所构成。当发生失效时,块备份可能在超组中的任何一组,这是因为对同名问题没有限制。将用来索引 cache 组的虚位可能产生的所有组号结合起来就组成了超组。在图 4-19b 中,两个虚地址共享一个物理地址,两组超组位可能有所不同,因此会指向不同的 cache 组。

由于同名问题,在同一个虚地址 cache 中,一个块的多个备份可能同时存在并被访问。对于只读的块,所有的备份都是一样的,尽管多个备份浪费了 cache 空间,但没有破坏—致性。对于可读写块,处理器可能会访问到旧的块。在图 4-19b 的例子中,如果在地址 PA 处的变量 X 是可写的,并且连续被 VA1 和 VA2 两个虚地址访问,如果 CPU 修改 VA2 对应的备份,则 cache 中会出现两个不一致的备份。这种情况下,除非给备份增加时间戳,否则无法记录最近访问的备份,即使有时间戳,每次 load 都要在整个超组中查找最近的备份也是不现实的。因此,不同别名对应的同一块在一个 cache 中不能存在多个备份。

别名必须进行动态检查。所有映射到同一组的同名都可以正确命中,因此,在命中时不需要别名检查。当将块加载到 cache 时,如果失效必须进行同名检查。cache 控制器必须检查超组中的所有组以找出同名,这个检查过程可以与取失效块并行执行,这一过程非常高效,因为它可以使用正常 cache 访问的索引硬件。如果块已经在其他 cache 中,但是使用的是别名,我们称这种失效为短失效 (short miss)。这种情况下,不需要从内存中取块,而只需移动 cache 内两个不同组的块。

当超组很大时,则检查整个超组是不现实的。另一种动态检查别名的方法是维护一个 cache 的反向表,它需要用物理地址而不是虚地址访问。一个常用的解决方法是基于可用物理地址访问的双重 cache 目录。理想情况下,可以假设一个全相联的双重目录,目录项数同 cache 目录项数相同,双重目录通过物理地址访问,并且包括一个指向 cache 行的反向指针。当 L1 cache 失效时,通过从 TLB 得到的物理地址访问双重目录。如果存在有效的指向 cache 的反向指针,则表明有同名存在。双重目录也可以是组相联的,且两个目录可能采用不同的组织方式。当一个新块被加载到 cache 中时,必须在两个目录中都找到空表项,这可能会导致 cache 中的两个块被换出 (成对换出),因为所有有效的 cache 块必须在双重目录中都有有效的反向指针。在双重目录中被换出的表项对应的 cache 块也要从 cache 中换出,这种做法在单处理器系统中代价很高,但在采用侦听协议的基于总线的多处理器中可以应用,这种系统通常需要双重目录来维护—致性。

假设 cache 之间采用包含关系,反向指针也可能在二级 cache 中维护。二级 cache 按照物理地址访问,因此,二级 cache 表项可以记录某块是否在 L1 cache 中,这种情况下,会包含一个

指向 L1 cache 行的反向指针。L1 cache 的失效会访问二级 cache，如果反向指针指向 L1 中的另一行，则 L2 cache 返回这个指针，L1 控制器执行短失效处理。这种短失效的延迟同 L1 失效 L2 命中的开销一样。

通过超组搜索的形式，硬件可以动态检测重名，包括双重目录或者 L2 中的反向指针，这种机制也称为反别名硬件（antialiasing hardware）。

最后，软件解决同名问题的方法称为页着色。页的颜色就是超组位的值，如果操作系统保证所有同名页有相同的颜色，则所有同名会索引到相同的组。

4.4.9 带虚标识的虚地址 cache

概念上，可以将 TLB 移到一级 cache 之后，这是解决虚存开销的一种有效方法，如图 4-20 所示。在这种结构中，L1 cache 通过虚拟地址位访问（不只是索引），L1 中的标识也是虚拟的。二级 cache 仍是按物理地址访问，位于 TLB 转换之后。因此唯一需要虚实地址转换的是出现 L1 失效时，TLB 失效仍然可以当作是处理器异常或通过 MMU 处理。这样做的主要优点是 TLB 不再处在访存关键路径上，并且可以做得很大，从而带来更高的命中率。

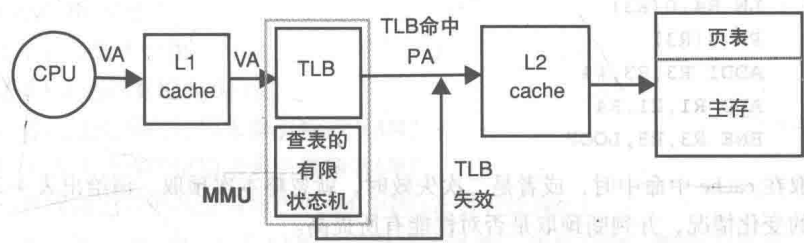


图 4-20 使用虚标识访问虚地址 cache

但是，这种方法在硬件和软件上都很难实现。在硬件层面，反别名硬件更加复杂，尽管带实标识的虚地址 cache 的基本解决方法仍然适用。此外，由于页表和 TLB 中的页表项的状态位在 cache 命中时不会被访问，因此必须在每个 L1 的 cache 行中都维护。这非常关键，尤其是 RWX 位。最后，软件必须熟悉 cache 内容，并且在很多情况下要换出部分 cache。

习题

4.1 本题涉及 cache 失效、结构冲突以及乱序处理器中的非阻塞 cache 行为。考虑如下的简单程序片段：

```
LOOP:    LW R4,0(R3)
        ADDI R3,R3,#4
        ADD R1,R1,R4
        BNE R3,R5,LOOP
```

- (a) 代码在如图 3-23 的 Tomasulo + 推测执行的机器上执行，ROB 有 8 项，并且一旦 ROB 满了，就停止分配，以免发生结构冲突。当最老的指令退出后又有一个新的 ROB 表项变为可用。数据 cache 是非阻塞的，并且有一个 R/W 访问端口，因此，所有的 cache 访问在达到访问端口时都是串行执行的。命中返回数据以及失效时查找 cache 都需要一个周期，R/W 端口在失效后返回 load 值时也处于忙碌状态。Miss 对应的延迟是 9 个周期，因此当命中时只需要 1 个周期，而出现 miss 时，则总共需要 10 个周期。一旦初次失效执行完成，对同一块的二次失效由 CDB 上的 cache 控制器来处理，每个周期可以处理一个失效。当 cache 失效时，CDB 上返回的数据的优先级低于其他指令返回的值。本题中假定 cache 大小为 32KB，采取直接映射，每个块大小 8 字节（2 个字），因此在本题程序中，一个初次失效后会有一次二次失效。假定 cache 初始为空，表 4-1 给出了前 5 个周期的执行状况。在 Dispatch 这一列，括号内给出了占用的

ROB 项，这个数字永远小于 8。请给出前 4 次迭代对应的表项变化情况，假设分支预测结果总是跳转。

表 4-1 带推测的 Tomasulo 算法（无预取）

		分发	发射	开始执行	完成执行	cache	CDB	retire	备注
I1	LW R4, 0(R3)	1(0)	2	(3)	(3)	(4)	(5)		初次失效
I2	ADDI R3, R3, #4	2(1)	4	(5)	(5)				CDB conflict with I1
I3	ADD R1, R1, R4	3(2)							
I4	BNE R3, R5, LOOP	4(3)							
I5	LW R4, 0(R3)	5(4)							二次失效

(b) 为了避免失效，向代码中插入新的非约束性预取指令。新的指令形式为 $PW\ d(R)$ ，其中 d 表示偏移量， R 表示基址。这条指令向 cache 中加载一个块，但不向 CDB 返回数据，也不需要 Retire，预取产生的所有异常都被丢弃。新的代码如下：

```
LOOP : LW R4, 0(R3)
        PW 8(R3)
        ADDI R3, R3, #4
        ADD R1, R1, R4
        BNE R3, R5, LOOP
```

当预取在 cache 中命中时，或者是二次失效时，就忽略本次预取。请给出表 4-2 中对应前 4 次迭代的变化情况，并判断预取是否对性能有所提高。

表 4-2 带推测的 Tomasulo 算法（带预取）

		分发	发射	开始执行	完成执行	cache	CDB	retire	备注
I1	LW R4, 0(R3)	1(0)	2	(3)	(3)	(4)	(5)		初次失效
I2	PW 8(R3)	2(1)	3	(4)	(4)	(5)			
I3	ADDI R3, R3, #4	3(2)	4	(5)	(5)				
I4	ADD R1, R1, R4	4(3)							
I5	BNE R3, R5, LOOP	5(4)							
I6	LW R4, 0(R3)								

4.2 本题中讨论设计支持虚存的结构并确定其大小。假定机器的虚地址是 42 位，并且配置有 256MB 物理内存。机器字大小为 64 位（8 字节），地址按字节编址并且按字节对齐。用下面的标记来表示一个 i 位的地址： $A_{i-1} \cdots A_2 A_1 A_0$ ，其中 A_{i-1} 是最高位， A_0 是最低位。虚地址表示为 $V_{41} \sim V_0$ ，物理地址表示为 $P_{27} \sim P_0$ 。

(a) 假定页表如下配置：

页大小：4KB

页表：三级页表，虚页号分为 3 部分，分别为 8 位、11 位、11 位

页表项：32 位（4 字节）

(1) 虚地址中的哪些位用于索引第一级页表？

(2) 虚地址中的哪些位用于索引最后一级页表？

(3) 每级页表的大小是多少（以字节为单位）？

(4) 每级页表项所能覆盖的虚地址空间是多少？

(b) 假定 TLB 如下配置：

TLB 大小：256 项

TLB 组织方式：两路组相联

- (1) 虚地址的哪些位用于索引 TLB?
- (2) 虚地址的哪些位用作 TLB 标识 (tag)?
- (3) TLB 的标识大小是多少?

(c) 假定二级 cache 如下配置:

cache 大小: 5MB
块大小: 64 字节
组大小: 每组 10 个 cache 行

二级 cache 用物理地址访问, 每个数据 RAM 列的宽度为 64 位 (一个字)。

- (1) 物理地址的哪些位用来索引标识 RAM?
- (2) 物理地址的哪些位用来索引数据 RAM?
- (3) 标识大小是多少?

(d) 一级 cache 配置如下:

一级 cache 为虚地址索引物理标识, 三路组相联
一级 cache 大小: 96KB
块大小: 16 字节
组大小: 每组 3 个块
数据 RAM 列的宽度为 64 位。

- (1) 虚地址的哪些位用来索引标识 RAM?
- (2) 虚地址的哪些位用来索引数据 RAM?
- (3) 物理地址的哪些位与标识 RAM 中的 tag 进行比较?
- (4) 在一级 cache 中, 由于同名的原因可能存在一致性问题, 解决方法之一是在每次失效时检查所有可能包含该块的组, 这种情况下请问总共需要检查多少个组?
- (5) 另一种解决同名问题的方法是页着色, 请问需要多少位来定义页的颜色?

4.3 本题主要涉及支持大块虚地址空间的页表结构。假定有一个 42 位虚地址空间、256MB 物理内存的机器, 页大小为 4KB, 页表项为 4 字节。页表可以有多种层次结构: 一级、二级或者三级。需要映射的虚地址空间由图 4-21 给出。内核地址空间不需要转换, 因为这部分的物理地址与虚地址相同。不过其他段的地址都需要转换。

- (a) 若只有一级页表, 页表大小是多少?
- (b) 假设采用二级页表。30 位虚页号分为 2 个 15 位, 则总共需要多少个页表? 页表的总大小为多少?
- (c) 假定采用三级页表, 将 30 位虚页号分为 3 个 10 位, 重做问题 (b)。

4.4 什么是伪 LRU 算法? 上网找一篇介绍伪 LRU 算法的文章, 以包含 4 个 cache 行的 cache 结构为例解释其工作过程, 并说明其相对于 LRU 算法的好处。

4.5 本题讨论当内存访问呈周期性变化时的 cache 映射和替换策略。在指令访问 (比如循环) 或跨步数数据访问时, 这种周期性的访存操作非常常见。

一级指令 cache 通常是直接映射的, 并不只因为直接映射在命中时开销小, 还因为它比组相联更适合处理循环。

假设 cache 有 4 行 (0, 1, 2, 3), 周期性地访问块地址 (0, 1, 2, 3, 4, 5)¹⁰, 上述标记表示按序访问块 0, 1, 2, 3, 4, 5 并重复 10 次。我们将失效分为冷失效、容量失效和冲突失效。容



图 4-21 虚地址空间映射

量失效基于采用 LRU 替换策略的全相联 (FA) cache 计算得到, cache 初始为空。

- 计算下列情形下的 cache 总失效次数: 直接映射, 采用 LRU 替换策略的 FA, 采用 FIFO 替换策略的 FA, 采用 LIFO (后进先出) 替换策略的 FA, 采用 LRU 的两路组相联 cache。
- 基于 (a) 的结果, 每种 cache 中, 冷失效、容量失效和冲突失效的次数各为多少?
- 考虑采用最优替换策略的 FA cache。尽管实际无法实现, 但是最优策略可以保证最高的命中率, 最优替换策略总是可以替换掉未来最久不会被访问的块。我们引入一个块距离的定义, 它表示块的当前访问和下一次访问之间还访问了多少其他的不同块, 因此, 在最优替换策略中, 我们总是替换块距离最大的块。假设所有块在访问序列的最后都额外再访问一次。如果有多个块的块距离相同, 那么就从中随机选择一个进行替换。这样的话可能会出现不同的替换序列, 但是失效率是相同的。请问如果 FA cache 采用 OPT 策略, 总共会产生多少次失效?
- 对 (a) 中的所有 cache, 计算其冲突失效次数。假定计算容量失效的 FA cache 是采用最优替换策略的 FA cache。

4.6 本题根据给定的访问序列, 比较不同的替换策略。考虑下面的访问序列, 每个字母代表一个块地址:

aabcaadeffefefefabgcaef

- 假设采用全相联的 4 行 cache, cache 初始为空, 计算采用下列替换策略下的失效率
 - LRU。
 - FIFO。
 - 伪 LRU (参见习题 4.4)。
 - 最不经常使用 (least frequently used, LFU)。LFU 替换策略中, 记录对每个块的访问次数, 替换访问次数最小的块。如果出现多个块访问次数相同, 则随机替换一个。本题中, 为了保证结果的一致性, 当出现访问次数相同时, 采用 LRU 替换一个。
 - 最优替换策略。
- 计算下列情况下每个 cache 的冲突失效次数。
 - 计算容量失效次数, 基于 LRU 替换策略的 FA cache。
 - 计算容量失效次数, 基于最优替换策略的 FA cache。

4.7 大多数科学计算应用包含矩阵运算, 最常见的矩阵运算是矩阵乘法

$$X = Y * Z$$

式中, X, Y, Z 都是 N 乘 N 的矩阵。

如果 X 的元素按行计算, L1 cache 太小, 无法放下整个 Z 矩阵, 那么计算 X 的每行时都要重新加载矩阵 Z , 这导致总的 cache 失效次数达到 N^3 。为了降低 cache 失效率, 编译器可以进行矩阵分块乘法。

假设全相联数据 cache 采用 LRU 替换策略, 矩阵大小为 256×256 。每个元素是一个双精度浮点数 (8 字节)。简单起见, 假设 cache 块大小为 8 字节, 并且操作数是对齐的。

- 计算数据 cache 的总失效次数, 假设数据 cache 大小为 128KB。为了计算失效率, 忽略程序中的整数访问, 只关心浮点访问, 同时计算总的操作次数 (浮点加法和浮点乘法)。
- 为了降低失效率, 一名博士生认为可以对算法进行重构, 这个学生认为将矩阵分块为 $N/k \times N/k$ (k 是 2 的幂) 的子矩阵相乘, 由于 cache 很容易放下多个子矩阵, 因此执行时间和失效率都会得到改进。

比如, 当 $k=2$ 时, 原矩阵相乘变为一组 $N/2 \times N/2$ 的矩阵乘法和加法。

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \times \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix};$$

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} Y_{11} \times Z_{11} + Y_{12} \times Z_{21} & Y_{11} \times Z_{12} + Y_{12} \times Z_{22} \\ Y_{21} \times Z_{11} + Y_{22} \times Z_{21} & Y_{21} \times Z_{12} + Y_{22} \times Z_{22} \end{bmatrix}$$

假设 cache 大小 128KB, 请给出可以将失效率降至最低的分块算法, 并计算总的失效次数, 同

时计算总操作次数（浮点加法和浮点乘法）。

- (c) 考虑由 cache 失效引起的芯片 I/O 次数以及总的计算操作次数，并判断这个博士生的做法是否可取。

4.8 考虑将内存中向量进行累加的简单程序，其代码如下：

```
LOOP : LW R4,0(R3)
        ADDI R3,R3,stridex4 /stridex4: 跨距乘4
        ADD R1,R1,R4
        BNE R3,R5,LOOP
```

跨距 (stride) 是指两个相邻向量元素之间的索引值的差。将它乘 4 可以得到相邻的下一个元素的地址。

我们分别在支持阻塞和非阻塞数据 cache 的五级流水线上评估循环执行效率。当遇到跳转的分支时清空取指和译码流水级。

本题中的 cache 采用直接映射，初始为空，块大小为 64KB。假设循环的迭代次数很大（向量元素多达数百万个）。

起初，假设 L1 失效的延迟为 30 个周期，记住 LW 指令先访问 cache，如果失效，则流水线停顿 30 个周期，然后继续。

- (a) 先考虑阻塞式 cache 的情况。计算循环每次迭代的平均执行时间（以时钟周期为单位），时间表示为以跨距为变量的函数。

- (b) 接下来我们采用一个简单的非阻塞 cache，在这种 cache 中，当预取失效尚未处理完成时仍然可以继续执行 load 命中。预取操作在流水线中的执行类似于 load 操作，区别在于：①预取不返回值；②即使 cache 预取出现失效，也不会阻塞流水线。在本题中，只要 load 出现失效，流水线就会停顿，一直到失效解决或者之前未完成的失效预取已经返回 cache，此时 load 操作可以提交退出。新的预取指令表示为 PW d(R) 的形式，其中 d 表示偏移量，R 表示基址。若预取出现初次失效，则 cache 控制器从下一级 cache 中取数据块，若预取命中或者是二次失效，则 cache 控制器放弃预取操作。此外，所有碰到异常的预取也将被放弃。

假定如下代码：

```
LOOP : LW R4,0(R3)
        PW stridex4(R3)
        ADDI R3,R3,stridex4
        ADD R1,R1,R4
        BNE R3,R5,LOOP
```

计算循环每次迭代的平均执行周期（以时钟周期为单位），时间表示为以跨距为变量的函数。

- (c) 如何能够确定什么时候预取是有效的？预取的有效性是否可以表示成由跨距、块大小以及失效延迟组成的函数，通常的平衡点在哪里？

多处理器系统

5.1 概述

自从计算机系统诞生以来,人们对性能的需求就成为计算机体系结构演变最重要的驱动力。特别是单一(串行)处理器核已经无法满足许多重要应用的需求时,并行体系结构技术应运而生。例子之一是用于计算机仿真的数值程序,这类程序从科学和工程的角度分析问题和解决问题,比如气候建模、天气预报、计算机辅助设计等。另一个例子是商业系统,这类系统中需要支持大量的独立查询操作,以适应信息时代日益增长的用户需求。多年来,促进并行体系结构发展的另一个原因是即将到来的技术壁垒可能会导致串行计算机的性能增长出现停滞。上述两个原因推动了多处理器架构的研究,并且在21世纪初,多处理器技术开始转向多核技术。

本章主要介绍多处理器系统的设计原则,集中讲述了两种不同的多处理器架构:共享内存多处理器系统和消息传递多处理器系统。这两种架构都使用了多处理器,通过增加处理器数量实现计算能力的线性增加。两者的区别在于:处理器交换数据的方式不同。在共享内存多处理器系统中,多个处理器共享相同的地址空间,而且通过在共享内存单元用常规的load和store指令实现处理器间的数据交换;而在消息传递多处理器系统中,各处理器有自己的(私有的)地址空间,它们通过交换消息实现数据通信。

要想充分理解多处理器系统庞大的设计空间,很重要的一点是先要抽象出提供给软件的接口。不管是什么样的并行体系结构模型,软件编程都必须显式地体现出并行性。能够并行的计算任务必须先被识别出来,然后均衡分配到各个处理器上执行。在这个过程中,必须深刻理解任务间的依赖和任务间进行结果传递的通信需求,这些统称为协调。同步和通信的原语是由底层结构支持并提供给软件调用的重要组成部分。我们首先分析共享内存多处理器和消息传递多处理器系统的编程模型,然后介绍这一过程中需要引入的原语。在共享内存多处理器系统中,其所提供的固有的共享内存地址空间能够直接支持通信机制,但仍需要显式地支持同步。另一方面,在消息传递系统中,需要为同步和通信都提供显式的原语:发送和接收原语。在本章中,将介绍这些原语、协议以及为加快消息处理的相关结构支持。

本章的大部分篇幅讲解共享内存多处理器的设计原则,因为这是当前主流的片上多处理器系统。共享内存多处理器的设计核心环节是内存系统以及如何在内存系统中高效地向软件传达语义。为此,文中对比了各种cache组织形式,介绍了cache一致性的基本概念。虽然同一个内存单元可能在系统的不同cache中都存有副本,但通过cache一致性的支持,就好像整个系统中只有一块存储器一样,数据始终保持一致。不过,要想做到性能较好同时复杂度又可接受,实现cache一致性是一项非常有挑战性的工作,本章将系统地讨论各种常见实现方案。

本章涵盖的主要内容如下:

- 5.2节主要概括介绍共享内存多处理器和消息传递多处理器的编程模型。
- 5.3节主要介绍消息传递原语的语义,支持这些语义所需的协议,以及支持加速消息处理的体系结构。

- 5.4 节主要介绍 cache 一致性的概念, 基于侦听的 cache 一致性协议设计, 以及 TLB 一致性策略。
- 5.5 节介绍可扩展的共享内存系统, 其重点在于可应用于大规模系统的 cache 一致性解决方案设计, 以及可挖掘局部性的页面映射软件技术。
- 5.6 节主要介绍全 cache 内存系统 (COMA) 的设计原则。

5.2 并行编程模型

并行编程模型 (parallel-programming model) 定义了如何使用高级编程语言来表达并行计算。直到转入多核架构以后, 并行计算机才获得足够的吸引力。并行编程模型通常在常用编程语言如 C/C++ 或者 Fortran 等的基础上进行简单的扩展——通常借助于应用编程接口 (API) 来实现。重要的语言扩展包括消息传递系统的消息传递接口 (MPI) 和共享内存系统的 OpenMP, 这两类模型吸引了人们对并程序设计的关注。由于并行编程模型具有快速和对多核计算机广泛适应性的特点, 因此其研究是一个热门话题, 预计更高效的编程模型仍将在未来几年内出现。我们使用了已经成熟的并行编程模型作为基础, 并以此为框架讨论需要提供给现有并行计算机的硬件/软件接口的原语, 本节的目的突出它们的关键特性, 以验证本章后面所涵盖的主流体系结构的原语实现过程。

无论是共享内存还是消息传递的并行计算机体系结构, 程序员或编译器必须能够表述出并行机制。具体包括两部分: 工作分配 (work partitioning) 和协调 (coordination)。工作分配方面, 可以并行执行的工作必须能被识别且分配给各处理器。本书中使用线程或进程说明代码运行在并行计算机的一个处理器 (或核) 中。线程通常是共享内存多处理器如多核微处理器中使用的术语, 而进程常用于基于消息交换实现通信的并行计算机。协调方面, 通过在不同处理器上运行线程并行地完成工作, 其结果必须与单处理器完成该工作的结果相同, 此时就需要协调各处理器。协调包括两个操作: 一个是并行线程彼此之间的同步, 另一个是线程间部分结果的通信, 类似地, 协调也要达到这样一种效果: 运行结果要和整个工作是在单处理器上运行时一样。下面我们进一步详细介绍工作分配和协调。

从一个串行程序开始, 程序员或编译器必须首先确认程序可以并行执行的部分。对于串行程序中先后执行的两个程序段 S_1 和 S_2 , 它们可以是函数或简单的连续循环迭代, 只在 S_1 和 S_2 相互独立的情况下, 即 S_1 产生的数据不会被 S_2 使用时, S_1 和 S_2 才可以并行执行。如果并行执行 S_1 和 S_2 获得的结果与先后执行它们所获得的结果是相同的, 那么我们称这段并行程序符合串行语义 (sequential semantic)。

挖掘多处理器中并行性的关键是找到程序中的独立代码片段。并行的一个常见形式是数据级并行。数据级并行意味着一组数据中不同数据单元的计算相互独立。比如矩阵乘法中, 结果矩阵的一个单元的计算与另一个单元的计算完全独立。一般而言, 数据级并行常存在于循环中, 循环也是并行编译器进行并行处理的主要目标之一。为了使并行编译器能够在循环中开发数据级并行, 必须确保循环中没有跨循环迭代的依赖, 即不同迭代过程的计算是相互独立的。当矩阵中的所有数据元素运行相同的计算时, 我们一般用 SPMD 并行来描述这种情况。SPMD 是指单程序多数据并行, 它表示在所有数据元素上运行相同的函数 (程序代码)。数据级并行有一个非常吸引人的特质: 问题规模越大, 能发现的并行性越多。

并行的另一种形式是函数级并行或称任务级并行, 不同处理器中执行相互独立的函数。函数级并行的常见形式存在于流媒体应用程序中, 在这些程序中不用函数轮流应用在数据流 (如视频帧) 上。如果有多个函数连续应用在一串数据流上, 这些函数可以组成一个函数流水 (function pipeline)。假定有 N 级流水线的话, 就最高可以获得 N 倍的加速。当然, 这两种并行

方法可以混合使用,以高效挖掘并行性。数据级并行可以用于函数流水的每个函数中,这样在两个级别都体现了并行性。

要实现协调、同步和通信,意味着共享内存系统中的线程需要通过内存交换信息,或者消息传递系统中的线程需要交换显式消息。不管是什么通信模型,很明显通信都会耗费一定的时间,从而影响并行计算机处理问题的速度。从体系结构的角度出发,支持高效的同步和通信实现非常重要。归结下来就是要理解如何消除共享内存系统或互连网络中的瓶颈。为了合理地权衡设计空间和开销,需要定义出硬件应该向软件提供哪些支持同步和通信的原语。我们会通过以下针对共享内存和消息传递系统设计的并行算法例子来识别这些原语。

我们考虑的并行计算问题如图 5-1 所示。算法完成两个矩阵 **A**、**B** 相乘,每个矩阵都是 N 行 N 列的(第 6 行)。此外,还计算出所有矩阵元素的和(第 7 行)。算法的输出是 N 行 N 列的结果矩阵 **C** 和矩阵 **C** 的所有元素的标量和。

```

1  sum = 0;
2  for (i=0,i<N,i++){
3      for (j=0,j<N,j++){
4          C[i,j] = 0;
5          for (k=0,k<N,k++){
6              C[i,j] = C[i,j] + A[i,k]*B[k,j];
7          sum += C[i,j];
8      }

```

图 5-1 两个矩阵相乘的串行算法的伪码

5.2.1 共享内存系统

矩阵乘算法(如图 5-1 所示)存在大量数据级并行,这是因为每个矩阵元素的计算与其他元素的计算是相互独立的,所以理论上所有矩阵元素都可以并行计算。然而,在处理器中创建一个线程会产生性能开销,如下面的例子所示,这些开销会减少并行程序实现的性能提升。

例 5.1 假设在 P 个核上执行两个矩阵相乘需要的时间为 T_{MM}/P , 开始一个新线程需要的时间为 T_{init} 。线程启动是顺序执行的。为使速度最大化,需要使用多少个核?

在 P 个核上矩阵相乘再加上线程启动的执行时间为:

$$T = P \times T_{init} + \frac{T_{MM}}{P}$$

最大化速度就是要找到使得执行时间 T 最小的核数 P , 也就是下式的解:

$$\frac{dT}{dP} = 0 \quad \text{或} \quad T_{init} - \frac{T_{MM}}{P^2} = 0$$

解得 P 的值为:

$$P = \sqrt{\frac{T_{MM}}{T_{init}}}$$

例如,在一个单处理器上执行矩阵乘法的时间(T_{MM})是线程初始化时间的 100 倍,那么使用 10 个核可以让计算获得最大速度。

并行粒度是指一个线程与其他线程并行执行的工作量大小。对于固定大小的问题,比如一个 $N \times N$ 大小的矩阵,并行粒度随着处理器数目的增多而减小。在图 5-1 的例子中,工作任务在 $nproc$ 个线程上进行分配,其中 $nproc = N/2^i$, $i = 0, 1, \dots, \log_2 N$, 共享内存系统中控制每

个线程的伪码如图 5-2 所示。

```

/* A, B, C, BAR, LV and sum are shared
/* All other variables are private
1a low = pid*N/nproc; /* pid=0...nproc-1
1b hi = low + N/nproc; /* identifies rows of A
1c mysum = 0; sum = 0;
2 for (i=low,i<hi,i++){
3     for (j=0,j<N,j++){
4         C[i,j] = 0;
5         for (k=0,k<N,k++){
6             C[i,j] = C[i,j] + A[i,k]*B[k,j];
7             mysum +=C[i,j];
8         }
9     BARRIER(BAR);
10    LOCK(LV);
11    sum += mysum;
12    UNLOCK(LV);

```

图 5-2 共享内存系统中并行算法的伪码

针对串行代码的所有改动在图中都被标为粗体。计算任务分配给 $nproc$ 个线程，第一个线程计算了第一个 $N/nproc$ 行元素的矩阵积，第二个线程计算了下一个 $N/nproc$ 行元素的矩阵积，以此类推。每个线程被分配了一个单独的标识 (pid)，其范围为 $[0, nproc - 1]$ 。第 1a 和 1b 行给出了使用 pid 号码的线程工作的行号范围。这组连续的行索引由变量 low 和 hi 定义，在三级循环嵌套的最外层循环中使用 (第 2 行)。

除了计算矩阵积，串行算法还计算结果矩阵中元素的和，一个难点是，在并行算法中需要注意避免因标量变量 sum 而造成的跨循环依赖。幸运的是，跨循环依赖可以通过让每个线程计算自己的行矩阵元素的和来避免。为此，每个线程维持一个私有变量，称为 $mysum$ ，用来累计局部的和。

在共享内存编程模型下，变量可以被定义为全局共享的。数组 A 、 B 、 C 以及标量变量 BAR 、 LV 和 sum 都是全局共享变量。这样的好处是矩阵可以被所有线程访问。因而，不需要再修改计算矩阵积的循环嵌套：每个线程可以读取输入矩阵 A 、 B ，并且将结果存入矩阵 C 中。不过，协调操作仍然是需要的，每个线程计算各自的局部和，这些局部和必须累计成为全局和。在第 1c 行，全局和与局部和都被初始化为 0。每个线程在第 10 ~ 12 行将它的局部和累加到全局和中。多个线程会分别将全局和读取到处理器寄存器中，将它们的局部和累加到寄存器中，最后将结果写回全局和变量中，在这个过程中一些局部和有被其他局部和覆盖的风险。因此局部和必须放在临界区内以串行的方式累加到全局和中。临界区的语义确保了任意时刻代码段最多只能被一个线程执行。实际上，在临界区，所有线程任意时刻只在全局和上累加它们各自的一个局部和，这样，最后结果就是所有局部和的正确和。临界区通常用锁机制来实现。

协调涉及的另一个问题是：什么时候让第一个线程将它的局部和加到全局和上才是安全的？当线程异步执行时，一个线程甚至可能在另一个线程开始执行之前就已经完成了它的局部矩阵积。后者在第 1c 行把和初始化为 0 之前，前者就计算完成它的局部和。显然，这将导致错误的结果。这时需要一个方法，使得在所有线程完成它们各自的矩阵乘法算法的部分之前，任何线程都不能进入临界区并进行和的更新。方法之一是采用栅栏同步 (barrier synchronization)。在语义上，栅栏同步会使得每个线程都等待，直到所有线程都到达该栅栏点才能推进。在上面的例子中，在第 9 行插入了一个栅栏同步。栅栏和锁一样都可以通过指令集架构中的 $load$ 指令、 $store$ 指令和读 - 修改 - 写指令来合成。我们将在第 7 章中详细讲述同步设计的实现

及其 ISA 支持。

总之,共享内存多处理器有一个很吸引人的特点,就是通信和同步操作可以通过 ISA 中的原语来实现。不过这也会导致 load 和 store 指令有时可能会产生非常长的访问延迟,并且底层存储系统的设计对性能的影响也至关重要。

下面介绍通过消息传递的机制来实现并行算法。

5.2.2 消息传递系统

共享内存系统是通过让所有线程共享同一地址空间来实现线程之间的协调,而消息传递系统却不同,每个线程或进程都有自己的地址空间。协调操作需要通过线程间显式地发送消息来实现。在任何由若干计算节点互连组成,并且每个节点都至少配备有一个处理器和一些内存的系统中,支持消息传递机制。消息传递在超大规模并行计算机系统(即由成百上千个计算节点构成的所谓计算集群系统)中是非常受欢迎的编程模式。线程间不共享任何数据有两个主要含义:首先,数据结构必须被显式地分配到私有地址空间;其次,线程执行的部分计算结果在最后必须进行收集。

矩阵乘法的消息传递版本如图 5-3 所示,图中给出了每个线程的代码。我们假设矩阵 A 和 B 初始时保存在一个计算节点(主节点)上,其线程编号为 0($\text{pid} = 0$)。如第 1b~1f 行所示,主节点将矩阵 A 的 N/nproc 行作为一块分配给其他线程。矩阵 A 只需要向每个线程发送大小为 myN 的部分矩阵(第 1e 行),而矩阵 B 则必须整体发送到所有线程中(第 1f 行)。

```

1a myN = N/nproc;
1b if(pid == 0)
1c   for(i=1; i<nproc;i++){
1d     k=i*N/nproc;
1e     SEND(&A[k][0],myN*N*sizeof(float),i,IN1);
1f     SEND(&B[0][0],N*N*sizeof(float),i,IN2);
1g   } else {
1h     RECV(&A[0][0],myN*N*sizeof(float),0,IN1);
1i     RECV(&B[0][0],N*N*sizeof(float),0,IN2);
1j   }
1k mysum = 0;
2   for (i=0,i<myN, i++)
3     for (j=0,j<N, j++){
4       C[i,j] = 0;
5       for (k=0,k<N, k++)
6         C[i,j] = C[i,j] + A[i,k]*B[k,j];
7       mysum += C[i,j];
8     }
9   if (pid == 0){
10    sum = mysum;
11    for(i = 1;i<nproc;i++){
12      RECV(&mysum,sizeof(float),i,SUM);
13      sum += mysum;
14    }
15    for(i=1; i<nproc;i++){
16      k=i*N/nproc;
17      RECV(&C[k][0],myN*N*sizeof(float),i,RES);
18    }
19  } else{
20    SEND(&mysum,sizeof(float),0,SUM);
21    SEND(&C[0][0],myN*N*sizeof(float),0,RES);
22  }

```

图 5-3 消息传递系统中并行算法的伪码

为了从一个节点向另一个节点拷贝数据，消息传递系统支持发送（SEND）和对应的接收（RECV）操作。SEND 的语义是指，从发送方的本地地址空间拷贝数据到接收方的一个缓冲区中。而 RECV 的语义是指，从缓冲区中拷贝数据到接收方的本地地址空间中。第 1h 行和第 1i 行的语句进行部分矩阵 **A** 和全部矩阵 **B** 的拷贝和接收。SEND/RECV 的参数从左至右依次是：本地数据结构的起始地址，消息的长度，接收者/发送者的 ID，一个同其他消息区分的标识。消息传递原语的详细语义将在 5.3 节中讲述。

对于主体计算部分（第 2~8 行），其代码与共享内存模型下的代码很像。主要区别在于，每个线程执行分配给它的由连续的 myN 行组成的部分任务，这些结果矩阵的本地部分稍后必须拷贝回主节点。和共享内存代码中一样，每个线程在私有标量变量 mysum 中计算部分和。

下面再考虑任务间的协调操作，在这一特定消息传递程序中，pid 为 0 的线程（主线程）负责计算总和，这可以通过让其他线程把部分和都发送给 pid=0 的线程来实现，这个 SEND 操作在第 20 行。这些部分和被主线程利用第 11~14 行的 for 循环收集起来并累加到 sum 中。各线程计算的部分矩阵积通过第 21 行的 SEND 发送回主线程，主线程执行的代码将部分结果复制到结果矩阵中，如第 15~18 行所示。

在该算法的共享内存实现版本中，通过特殊同步原语避免了数据竞争。在消息传递实现版本中，同步操作隐含在消息传递原语中。SEND 和 RECV 操作主要分两种：同步类型和异步类型。同步的 SEND 和 RECV 会阻塞进程直到双方通知对方消息已经完成交换。假设图 5-3 中算法实现的信息交换是同步的，那么代码中就不需要使用其他同步机制来避免数据竞争了。

注意，在消息传递系统中数据必须被显式地分配到各个参与计算的线程的存储区中，并且线程之间必须交换部分结果。SEND 和 RECV 原语的实现通常需要硬件和软件功能相结合。在 5.3 节中，我们将仔细研究消息传递系统的结构。

5.3 基于消息传递的多处理器系统

消息传递多处理器系统的核心是，参与节点之间显式地进行消息交换，因此，只有参与交换的节点才需要放在一起，这也使得这类系统具有良好的可扩展性。实际上，消息传递已经在包含数千个节点的大型计算机集群中成功实现。本节讲述包含处理单元和存储的节点之间的通信协议以及对消息传递的支持。消息传递系统可以基于桌面计算机集群很容易建立起来，也可以集成到共享内存的系统上。

5.3.1 消息传递原语

在消息传递系统中，消息交换是同步和通信的基本原语。消息传递原语在语义上不尽相同，我们从下面一个简单例子开始介绍同步消息传递原语。

例 5.2 在如下使用消息传递的程序中，进程 P1 和 P2 进行消息交换。当使用同步 SEND 和 RECV 原语时，打印结果是什么？

Code for process P1:

```
A=10;  
SEND(&A,sizeof(A),P2,SEND.A);  
A=A+1;  
RECV(&C,sizeof(C),P2,SEND.B);  
printf(C);
```

Code for process P2:

```
B=5;  
RECV(&B,sizeof(B),P1,SEND.A);  
B=B+1;  
SEND(&B,sizeof(B),P1,SEND.B);
```

一组对应的 SEND 和 RECV 原语本质上是在发送方和接收方之间搭建起了一条通信路径，这样发送方的某个预设好的地址空间中的数据就可以被拷贝到接收方的某个指定地址空间中。

在本例中, 有两组匹配的 SEND/RECV 对。在第一对中, SEND 的参数指出了本地地址空间 (&A)、消息长度 (sizeof(A))、接收方的 ID (P2), 以及将消息与其他 P1 和 P2 之间的消息进行区分的标识 (tag)。同样对应的 RECV 指明了接收方的本地地址空间 (&B)、B 的大小 (需要与 SEND 中的大小相同)、发送方的 ID (P1), 以及与对应 SEND 相同的 tag。由于 P1 和 P2 异步执行, SEND/RECV 协议的问题在于当 SEND 语句执行时, 要保证 A 的内容可以被正确复制到 B, 这需要发生在 P2 从 RECV 返回后执行下一条语句之前, 因为此时将真正改变 B 的内容。

在同步消息传递中, 发送方在接收方收到消息之前必须被阻塞。同样接收方在消息可用并且被拷贝到本地某个指定的地址空间之前也必须被阻塞。回到上面的例子, SEND 和 RECV 的同步机制保证了当 SEND 语句执行时 A 的内容 ($A = 10$) 被传送给 P2, 而不是执行 SEND 之后的值 ($A = 11$)。同样, 如果 RECV 是同步的, P2 在消息 (10) 被拷贝到本地变量 B 之前是被阻塞的。因此, 当 P2 发送包含 B 内容的消息给 P1 时, B 的内容是 11, 就好像在第一个 RECV 语句之后执行加 1 操作。接着 P1 将值拷贝在本地变量 C 中, 然后进行输出。

同步消息传递原语的一个好处是, 我们可以推测出程序执行的结果, 因为原语强制进行了同步。不过, 同步消息交换也有两个缺点: 一个是容易导致死锁, 另一个是不允许计算与通信重叠, 发送方必须等到接收方收到消息后才能继续执行。下面先来看一下死锁的情况。

例 5.3 考虑下面的同步消息传递程序, 为什么会出现死锁?

Code for process P1:

```
A = 10;
SEND(&A, sizeof(A), P2, SEND_A);
RECV(&B, sizeof(B), P2, SEND_B);
```

Code for process P2:

```
B = 5;
SEND(&B, sizeof(B), P1, SEND_B);
RECV(&A, sizeof(B), P1, SEND_A);
```

在同步消息交换下, 发送方和接收方都会阻塞, 直到消息传输结束, 也就是说一组匹配的 SEND/RECV 必须都执行完了才能往前推进。这段代码的问题在于, P1 和 P2 都阻塞在它们的 SEND 语句, 等待匹配的 RECV 执行, 但两者都无法执行各自的 RECV 语句, 因此程序发生了死锁。这段简单的代码可以通过交换 P1 或 P2 代码中 SEND/RECV 的顺序来打破死锁。

第二个同步消息传递的缺点是它将同步和通信结合在一起。同步和通信都是长延迟操作, 将它们封装在一个原语中会导致性能下降。将两者分开可以让发送方在消息传递的过程中做一些有用的工作。异步消息传递原语就是这样做的。在介绍之前, 先通过下面的例子来说明这样做的可能性。

例 5.4 考虑如下的消息传递程序。假设 SEND 需要 1000 个时钟周期来执行, 并且与之不相关的计算操作需要 500 个周期。假设采用异步 SEND 和同步 RECV 原语, 执行下面的消息传递程序需要多长时间?

Code for process P1:

```
A=10;
SEND(&A, sizeof(A), P2, SEND_A);
<UNRELATED COMPUTATION;>
RECV(&B, sizeof(B), P2, SEND_B);
```

Code for process P2:

```
B=5;
SEND(&B, sizeof(B), P1, SEND_B);
<UNRELATED COMPUTATION;>
RECV(&A, sizeof(B), P1, SEND_A);
```

在异步消息传递情况下, P1 发送消息, 然后在执行 SEND 之后继续执行下面的代码, 而不是等待接收方把消息拷贝到它的地址空间中。本例中, SEND 后的代码“不相关”, 是指这些代码不会修改 A 的内容, 也不需要 B 的值。在 P2 的结尾处, 同样 SEND 后的代码既不影响 B 的内容, 也不需要 A 的值。这样的话, SEND 的延迟有一部分可以被本地的计算所覆盖掉, 正如图 5-4 中的时间图所示。由于消息传递需要 1000 个时钟周期, 不相关计算需要 500 个周期,

消息传递的 500 个周期被有用工作重叠，所以执行这段代码需要 1000 个时钟周期。

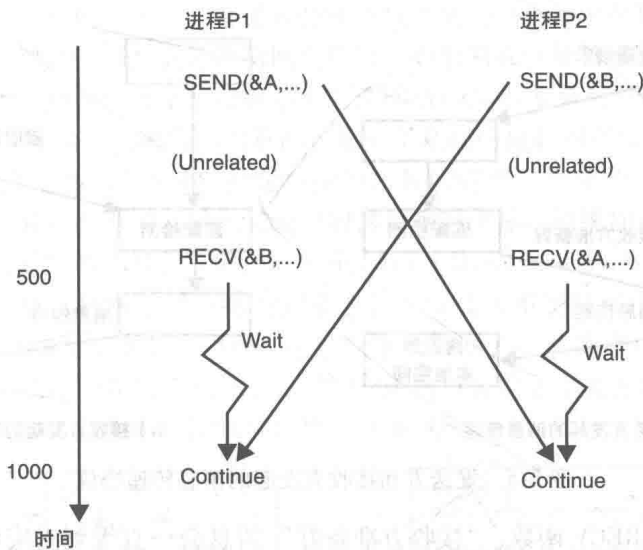


图 5-4 异步消息交换中计算与通信重叠

在异步消息传递中，发送方不用等到本地存储区中的数据被拷贝走就可以继续执行。在例 5.4 的代码中，由于消息传递与代码的执行是异步的，有可能出现当 A 的内容被拷贝时，它的值与 SEND 语句被执行时的值不一致。因此有两种异步消息传递原语：阻塞（blocking）和非阻塞（non-blocking）。接下来介绍这两种原语的区别。

当组成消息的本地数据在某个地方被缓存并且不会被发送进程的执行所改变时，阻塞的异步 SEND 会将控制权交给发送进程。同样阻塞的 RECV 直到消息被拷贝到接收进程的本地存储空间后，才会把控制权交回给接收进程。这与非阻塞的异步消息传递原语正好相反，非阻塞的异步传递会将控制权立刻返回给发送和接收进程，数据交换是在后台完成的。有的消息传递系统会提供探针函数（probe function）来检测消息传输的状态。探针函数可以检查数据是否从发送方的本地地址空间拷贝到缓冲区，或者，对称地，检查是否被拷贝到接收方的本地地址空间，以避免非阻塞异步消息传递导致的数据覆盖。

5.3.2 消息传递协议

在仔细分析了一对匹配的 SEND/RECV 原语在消息传递过程中的动作（包括同步的和异步的）之后，我们先考虑一个同步消息传输。当发送方执行 SEND 时，需要和接收方同步，确保它执行到了对应的 RECV。这时，包含从发送方本地地址空间拷贝来的数据的消息被发送到接收方，接收方再将消息拷贝回自己本地指定的地址空间。

图 5-5 给出了这种传输的一种实现协议。

在图 5-5a 中，给出的是发送方发起的同步消息传递协议。当发送进程执行 SEND 函数，发送给接收方一个消息询问接收方是否准备好（图 5-5a 中的“发送请求”）。接收节点为了确认自己是否准备好这次消息传输，会查询匹配表，这个表记录了所有之前执行的包括该进程 ID 和标识的 RECV 原语的状态。查表结果有两种情况：（1）对应的 RECV 已经执行过了；（2）对应的 RECV 还没执行。如果接收进程已经执行过对应的 RECV，表中就会出现一次匹配，并且接收节点会通知发送方，它准备好消息传输了（图 5-5a 中的“接收方准备好”）。发送方会将消息发送给接收方，然后接收方将消息拷贝到特定的本地地址空间。与之相反，如果

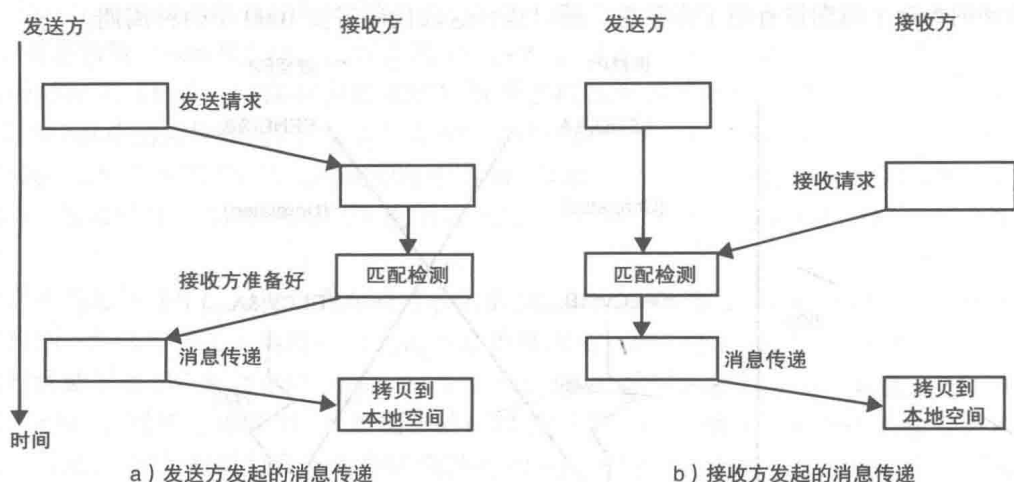


图 5-5 发送方和接收方发起的消息传递协议

接收节点还没有执行 RECV 函数，“接收方准备好”消息会一直等到对应的 RECV 函数被执行才发送，同时更新匹配表。注意这种协议需要 3 个阶段才能发送一次消息。

有人可能会问是否可以减少所需的阶段数。图 5-5b 给出了由接收方发起的两阶段同步消息传递协议。不同于发送方发起协议中匹配表由接收方维护的策略，在接收方发起的协议中，匹配表由发送方来维护。当接收方执行了 RECV 函数，会发送给发送节点一个消息（图 5-5b 中的“接收请求”），发送节点查询匹配表。同图 5-5a 给出的发送方发起的协议相同，查表结果可能会出现两种情况：一种是成功匹配后进行消息传递，另一种是接收进程会一直阻塞直到发送方对应的 SEND 函数执行。

在发送方或接收方发起的消息传递协议中，假设消息传递原语是同步的。这就是说，发送进程和接收进程都会一直阻塞直到传输完成。现在来考虑一下异步消息传递的情况。在阻塞的异步消息传递中，发送方可以越过 SEND 函数继续执行，前提是消息已经被放入缓冲区，这样本地数据结构的改变也不会影响消息的内容。需注意的是，在发送进程越过 SEND 继续执行后续语句之前，必须确保预留的缓存空间足以保存发送的消息。这个缓冲空间既可以设在发送方也可以设在接收方。将缓冲区设在接收方带来的一个问题是，发送方必须发送消息来查询是否有足够大小的缓冲区，在没有足够缓冲区之前，发送进程会被阻塞。当然，最好的情况是对应的 RECV 已经被执行，这样可以马上开始消息传输。但在最坏的情况下，发送方会一直阻塞直到有足够的缓冲区，这可能需要很长的时间，因为此时接收进程可能正在从其他很多节点接收消息。另一种替换策略是将缓冲区设在发送方，这种情况下，如图 5-5a 所示的发送方发起的传递协议会有所不同，本地数据结构的拷贝和发送请求给接收方是并行的。一旦拷贝完成，发送方就可以越过 SEND 函数继续执行。

5.3.3 消息传递协议的硬件支持

消息传递系统可以构建在通过互连网络连接的节点上，协议可以在支持它的底层网络事务原语的基础上构建。这些原子的网络事务由通用互连网络提供。因此，最基本的硬件支持是现成的。除此之外的硬件支持主要用于降低节点间消息传递的延迟，或者将消息传递协议的处理过程从计算处理器中分离，以直接提高计算速度。即使在异步协议下，发送方和接收方也可以不用显式地等待消息传递的完成，但更短的消息延迟仍然可以缩短执行时间，因为它可以带来更高的消息传输吞吐率。

正如我们在 5.3.2 节看到的那样，消息传递协议需要通过网络传递发送方地址空间中的一些数据的副本。如果没有硬件支持，发送方必须将数据从用户地址空间拷贝到本地系统存储区中，然后再从这个存储区中将数据传送给网络接口。如果只有计算处理器的硬件支持，传输延迟可能会很高，因为处理器的速度可能跟不上互连网络的注入速率（injection rate）。对于接收方也是一样的，图 5-6a 给出了相应的示意图，显示了从网络接口到系统区的输入缓冲区（1 和 2）以及从系统区到用户区（3 和 4）的拷贝操作。我们需要一个硬件结构支持，可以将数据从用户地址空间拷贝到系统空间，再从系统空间拷贝到网络接口的缓冲区中。同样，接收方的拷贝操作也需要相应的硬件支持。实际上，一个 DMA（Direct Memory Access）引擎就可以完成这些工作，它也是将数据从网络接口拷贝到系统区，以及从系统区再拷贝到用户区的常用机制，如图 5-6a 所示（细箭头部分）。DMA 引擎程序的编写，需要数据的首地址、目的存储单元地址以及存储区中连续的存储单元大小。然后它就可以分别在发送方将数据从系统区拷贝到网络接口，以及在接收方将数据从网络接口拷贝到系统区。

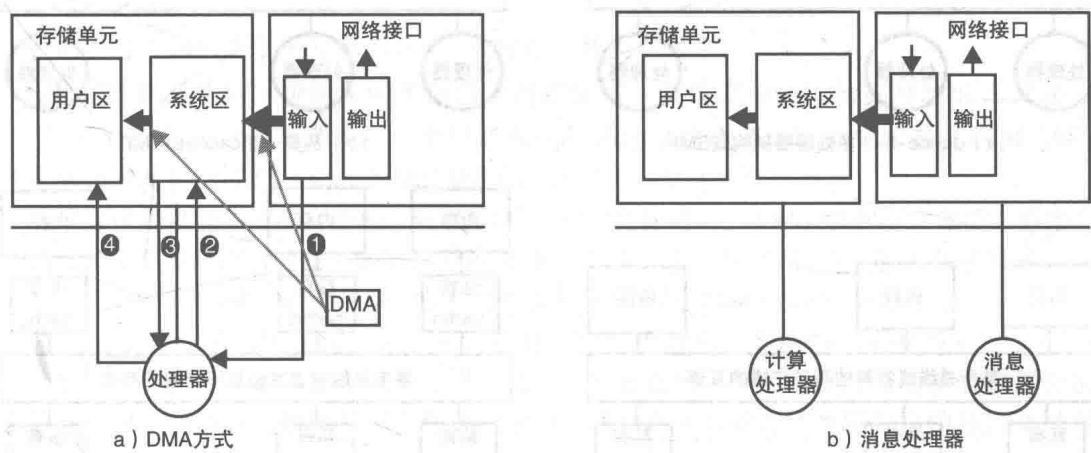


图 5-6 消息传递的硬件支持

输入的数据由操作系统层的程序处理，它会启动一个 DMA 引擎，将输入的数据存到系统区中。这样做有个缺点，就是必须把消息从系统层地址空间拷贝到用户层地址空间，这会增加开销。如果消息可以直接发送到用户空间，那么就可以存入匹配表的地址中，从而避免从系统空间到用户空间的拷贝。为此，可以在消息传递协议的最底层将系统层消息和用户层消息区分开，这样一来，当网络接口接收到消息后，会确定将消息直接传送到系统层存储空间还是用户层存储空间。

不同于减少消息传输的开销，另一种改进方法是将资源处理的工作交给进行消息传输的任务，这样，这些任务就被从计算处理器（CP）上卸载下来了。我们已经看到，DMA 引擎可以用于将数据在存储区和网络接口之间来回拷贝。一个通用的消息处理器（MP）也可以完成一些更高层的功能，如匹配、打包数据、发送确认消息等，就像图 5-6b 给出的那样。在某些实现中，多处理器系统中的某个计算处理器可以用来进行消息处理工作，这是一种非常值得一试的策略，因为在多核芯片中处理器或线程资源是非常丰富的。此外，还有一些实现将专用的消息处理器紧密耦合在网络接口上来处理协议事务。

5.4 基于总线的共享内存系统

本节讲述小规模共享内存多处理器系统的设计。从 cache 组织开始，然后讲解 cache 一致性的基本问题。本节的核心在于大量的 cache 一致性协议设计原则，我们暂时先假定系统中

有一个基于广播的互连机制，比如总线。

5.4.1 多处理器 cache 组织

基于总线的多处理器系统的组织方案如图 5-7 所示。图 5-7a 给出了基本的“dance-hall”组织结构，或者称为对称多处理器（SMP）系统。其典型特征是所有的处理器在连线的一边，所有的内存模块在另一边，这样，所有处理器的访存时间就是相同的。本节中我们假定互连都是基于总线的（或者任何基于广播的互连），在 5.5 节我们将会介绍任意拓扑结构的互连方式。

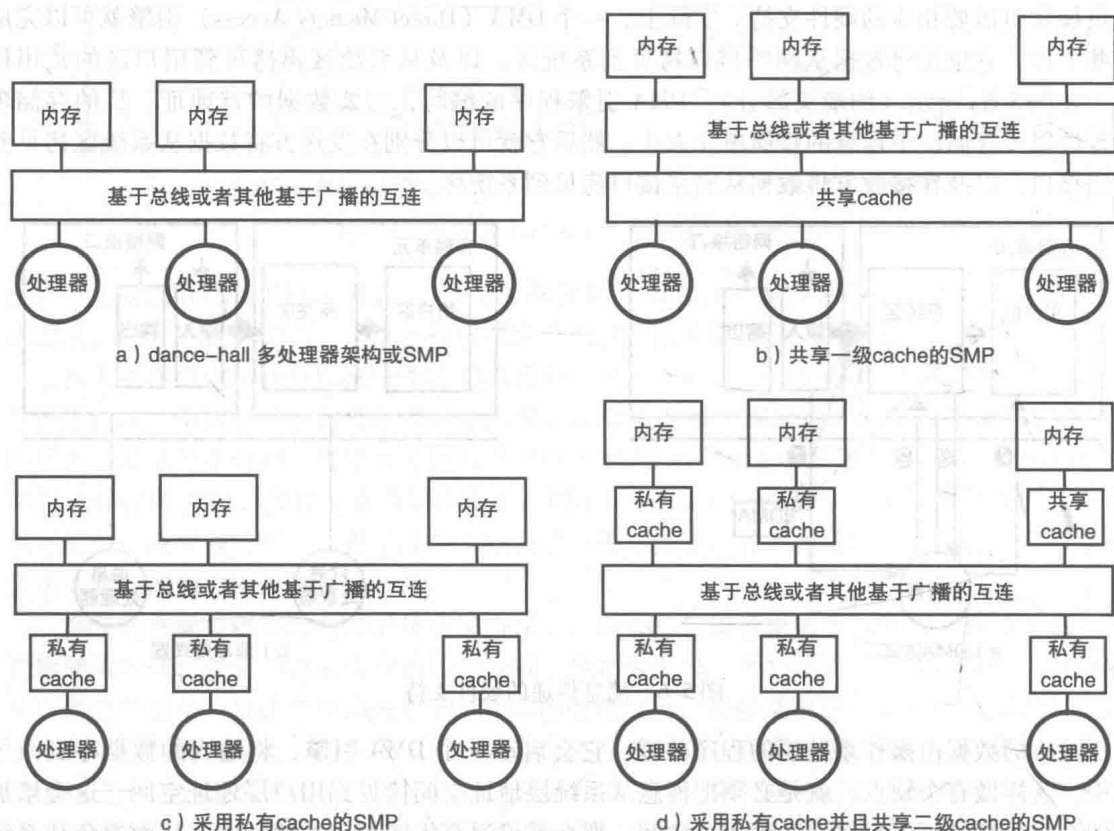


图 5-7 基于总线的多处理器组织方案

图 5-7a 给出的只是一个概念性的模型而不是真正实现的模型。由于存储层次缺少 cache，该模型并不实用。cache 存储层次对于单处理器系统至关重要，它可以弥补处理器和内存之间的速度差，它在多处理器中的重要性又进一步扩大到两个方面：首先，互连增加了存储器延迟；其次，由于存储器和互连网络是共享资源，其带宽就变得非常稀缺，而 cache 层次可以减少访存开销，并且降低存储器和互连网络的压力。因此，cache 层次结构是多处理器设计的一个基本考虑。下面我们来讨论 cache 结构的选择以及如何取舍。

先考虑一级 cache，我们需要决定选择共享的（shared）还是私有的（private）cache 组织结构。顾名思义，处理器私有的 cache 只能被该处理器访问，而共享的 cache 是一组处理器共用的，能被组内的所有处理器访问。图 5-7b 给出了在处理器和互连网络之间插入共享 cache 的结构。这种组织结构的好处是处理器之间的共享架构使得 cache 利用率更高。比如，两个处理器需要访问同一个 cache 块，那么就可以对其进行共享，而不用将数据拷贝两份分别放到各自私有的 cache 中。此外，如果第二次访问时数据仍在 cache 中，那么两次访问只有一个会失效。共享 cache 组织方式的一个缺点是，处理器和 cache 之间的互连网络会增加访存关键路径上处

理器到一级 cache 的延迟。因此,一级共享 cache 只适用于处理器数较少的结构。这也导致了如图 5-7c 所示结构的出现,这种结构中,私有 cache 和处理器在互连网络的同一边。注意我们必须支持私有 cache 这种结构,因为现在所有的处理器都有片上 cache。

图 5-7b 和 c 中的结构截然不同,此外还有一些混合组织方式。比如,图 5-7d 中的多处理器组织结构,既有在处理器同一侧的私有 cache,也有在存储模块一侧的共享 cache。使用二级 cache 的目的是缩小一级 cache 和内存之间的访存延迟。实际上,现在市面上的片上多处理器,其结构同图 5-7d 所示的结构十分相似,我们将在第 8 章进行介绍。

在本节剩余部分,我们考虑如图 5-7c 所示的一级私有 cache 结构的设计空间。多个私有 cache 的存在,意味着同一个内存块可能会被拷贝到不同的私有 cache 中。这些副本之间必须保持一致,即两个或多个处理器在任何时间访问同一地址的数据,返回的结果必须相同,这是 cache 一致性问题的精髓。这一问题设计共享内存多处理器时所需要考虑的最重要问题之一。下面我们将讲述解决 cache 一致性问题的通用方法。

cache 一致性

我们通过下面的例子来介绍 cache 一致性的一些重要概念。

例 5.5 考虑图 5-7a 中简单的多处理器组织结构。假设 P_1 到 P_N 轮流对地址 A 发起写操作,用 W_i 表示 P_i 发送的写请求。一个写请求执行的结果是改变内存中对应单元的值。那么,当所有写请求都执行完后 A 单元的值应该是什么?

结果有 N 种可能的情况,例如, W_3 是最后执行完的写请求, A 单元将保存它的值。但是,这些访存可能存在多种排列组合。比如,考虑 $W_N W_{N-1} \cdots W_1$ 这一写序列, W_1 是这个序列最后一次写。 A 的值就是 W_1 的值,我们称 W_1 是 A 的最后全局写入值 (last globally written value)。

像图 5-7a 所示的组织结构,每个单元只有一份拷贝内容,当新的写操作执行时,旧值就被替换掉了。任何一个处理器在一个写操作之后,并且在另一个写请求之前的读请求都会读到写操作的结果。当所有处理器都不会读到旧值时,我们称所有的处理器所看到的最后全局写入值是一致的。这个概念便于我们理解下面对于 cache 一致性的非正式定义。

定义 5.1 (cache 一致性) 一个 cache 系统是一致的,当且仅当在任何时间点,所有的处理器所看到的任意位置的最后全局写入值都是一致的。

第 7 章将给出正式的严格限制条件来保证 cache 一致性。这种直观上大家都期望的特性在系统对同一单元存在多个备份的情况下就会失效。如图 5-7c 和 d 那样的系统,存在私有 cache,任何物理地址单元都可能存在多个备份。我们假设私有 cache 采用写回或写穿透的方式,讨论 cache 一致性的问题。

在写回方式下,写操作只会更新本地 cache 的备份。这样的话,两个处理器对同一地址执行写操作,就会导致对最后全局写入值的不一致结果,之后的读操作也会返回不同的值。另一方面,采用写穿透方式,内存单元的本地备份和内存中的值会在一次写操作中同时被更新。这样,该单元的内存备份,对于那些本地没有备份的处理器来说,保持了对最后全局写入值的一致结果。然而,对于那些本地保存了拷贝的处理器来说,其所看到的结果还是非一致的。从这些例子可以看出,对单处理器设计的 cache 结构无法保证 cache 的一致性。我们必须扩展基本的 cache 结构,使其能够保证对最后全局写入值的一致结果。本部分学习的重要目标是理解 cache 结构设计所需要的设计空间,以及理解各种用来解决 cache 一致性问题的协议。

5.4.2 一个简单的侦听 cache 协议

为了介绍侦听 cache 协议的设计概念,我们先来简化一下对于 cache 的一些假设。为了简

便起见，假设每个处理器的 cache 是阻塞式写穿透写分配的，多处理器组织结构如图 5-7c 所示。“阻塞”是指处理器在 cache 请求结束前一直被挂起。“写分配”指如果写访问的数据块不在 cache 中，要先将这一块取到 cache 中。cache 被连接到互连总线，这个总线同一时刻只能处理一个总线事务。也就是说，一旦 cache 控制器获得了总线访问权，它就会一直独占总线，直到事务处理完毕。

第一种解决 cache 一致性的协议很简单，它的大致动作如下：处理器的读请求同单处理器系统处理方式相同，对于写请求，则需要额外的动作。对于在一个 cache 中存在的块上写入，内存像单处理器系统那样更新。但是，在所有保存该块副本的 cache 中，该数据块被无效掉，比如，置有效位为 0。如果该块不在 cache 中，首先将这一块取到 cache 中，然后更新这一块在内存中相应位置的值，并且无效掉其他 cache 中保存的副本。

新的操作是无效掉所有 cache 中的某个特定块。在一个基于总线或其他广播方式的互连网络的多处理器组织结构中，这个过程可以被简化，因为所有的 cache 可以侦听或探查所有的总线请求，并且选择同它们有关的请求。采用这类协议的方法被称作侦听 cache 协议。

简单协议的硬件结构

现在来看实现这种简单侦听 cache 协议所需的硬件结构。图 5-8 给出了写穿透 cache 的组织细节以及它的写缓冲。

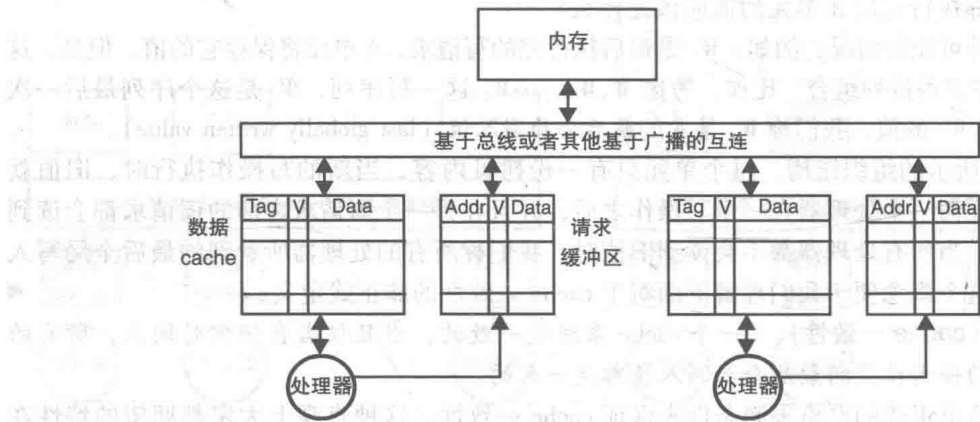


图 5-8 简单侦听 cache 协议的硬件结构

图 5-8 中的两个 cache 都由两部分组成：一个数据 cache 和一个请求缓冲区，这个缓冲区是单处理器系统的写缓冲。数据 cache 中的每一项都有一个状态位（state bit），这在写穿透 cache 中只是用来区别有效项和无效项。为了实现简单的侦听 cache 协议，cache 不仅要响应来自处理器的读和写请求，还要响应来自总线的无效请求。为了理解简单协议通过这些硬件实现的动作，我们来仔细看一下写失效和读失效请求的动作。正如前面提到的，读命中的处理同单处理器的 cache 一样，我们不再解释。我们将初始化协议的 cache 叫作本地 cache，将其他协议涉及的 cache 称作远程 cache。

简单协议的行为特征

我们还记得，在单处理器环境中，读失效（read-miss）请求可以绕过写缓冲中还未执行完的写请求，只要其地址同写缓冲中未执行写请求的地址不一样就可以。同样，如果它们地址匹配，一个读失效的请求可以立即返回未执行的写请求的值。然而，这些优化方案在多处理器系统中通常都不允许，我们将在第 7 章进行详细的解释。

出现读失效时，读失效请求会插入到请求缓冲区下一个空闲项中，并且将 V 位（有效位）

设置为有效。当读请求到达请求缓冲队列头时，本地 cache 控制器会申请总线控制权，当申请成功后，会向总线发送总线读请求（BusRd），并且返回内存块的一份拷贝。

对于写命中（write hit），写请求的值以及地址将被插入到请求缓冲中。一旦写请求到了缓冲队列头，控制器就将请求独占总线。一旦请求允许，就会发送总线写请求（BusWrite）。这个写请求会更新内存，并对所有远程 cache 进行标识检查。在远程 cache 中，只要有同 BusWrite 请求匹配的拷贝数据，其对应的 V 位就将清空，从而使相应的块备份无效。在将总线控制器释放之前，最后一步是用写入值更新本地 cache。

最后，对于写失效（write miss）。在写分配策略下，在写操作发生之前数据块会被取到 cache 中，写失效请求会被插入到请求缓冲中。当请求到了缓冲队列头时，先获得总线控制权，然后发出总线独占读请求（BusRdX）。除了将数据块从内存取到本地 cache 中，还要对所有远程 cache 进行标识检查，任何匹配的远程 cache 块都将被无效掉。此外，像 BusWrite 事务一样，内存拷贝也将更新。非写分配策略下的 cache 写失效处理有些不同，BusRdX 事务可以用一个简单的 BusWrite 取代。

正如之前提到的，本地 cache 需要等到总线事务完成之后才更新写的值。我们将在第 7 章看到，这对于保证正确性非常重要。从 cache 一致性的非正式定义可以获得一个直观的解释，一旦总线事务完成，新值就会被认为是“最后全局写入值”。因此，在所有处理器看到新值之前，不能把新值写给本地处理器。回到 cache 一致性的非正式定义，当写请求被执行后，从任何一个处理器来看，值都是“最后全局写入值”，因为对所有处理器来说，对这个值都是一致的。

现在，考虑当两个处理器同时访问同一数据块时可能带来的问题，下面举例说明。

例 5.6 假设两个处理器 P_1 和 P_2 同时对地址 A 发出写请求，并且 A 对应数据块在两个 cache 中都有副本，那么协议是怎么处理的？

两个处理器都向它们自己的请求缓冲插入写请求，并且两个控制器都会尝试获取总线。如果 P_2 的总线请求先被允许，其对应的 BusWrite 请求会无效掉 P_1 cache 中的数据块。当 P_1 最终获得总线控制权时，如果采用写分配策略，它需要发送 BusRdX 请求，而不是 BusWrite 请求。这是因为现在的数据块已经过期了，必须从内存中读取新块。这个例子说明，由于 cache 间相互干扰从而引入了一个非常重要的设计问题。一种解决方法是取消 P_1 的写请求，并且尝试发送 BusRdX 请求。另一方面，在非写分配策略中，不管哪一个 cache 控制器获得了控制权，正确的动作都是保持 BusWrite 请求。这是 cache 一致性协议设计时需要处理的众多微妙问题中的一个例子。我们稍后会讲到其他问题，这些问题都是由于目前对于同一数据块的并发请求时的竞争条件所引起的。

在支持侦听 cache 协议的设计中，标识查看必须对所有请求以及所有 cache 都执行，这又引入了一个新的设计问题——cache 控制器不仅要对处理器的请求进行响应，还要响应总线的请求，这可能会导致拥塞。总线请求引起的冲突可能会导致处理器被锁住，从而引起更高的访存延迟。缓解这种情况的办法之一是复制多个标识存储，并且将处理器一侧和总线一侧的标识存储分开。只要处理器或总线的请求不改变标识或 V（有效）位，两个标识存储中的检查就仍是独立的。不过，如果标识存储拷贝中的某一项发生变化，那么两个标识存储都需要更新从而保证一致性。值得庆幸的是，这种更新只发生在发送总线请求或接收到总线请求时，并且只是在一部分情况下需要更新标识存储拷贝。

描述行为规范的状态转换图

刻画 cache 协议高层行为特征的方法是状态转换图（state-transition diagram）。简单协议的

状态转换图如图 5-9 所示。对于 cache 控制器的任意一个给定输入，图中给出了对应 cache 块拷贝的下一个状态。除了状态转换外，图中还给出了状态转

换的结果将导致哪种总线事务。此外，每个状态还控制了处理器读或写请求是可以本地完成还是需要发起一个 cache 协议事务。表 5-1 列出了本章中所讲述的 cache 协议对应的所有 cache 控制器的输入和输出。

图 5-9 给出了协议的两种状态：有效或无效。如果副本块是无效状态，则处理器读（processor read，PrRd）请求会触发一个 BusRd 请求。这一转换在状态转换图中用 PrRd/BusRd 表示。对无效块的处理器写（processot write，PrWr）操作会向总线发送 BusRdX 请

求。如果块是有效的，处理器读不会导致状态改变，也不会引起任何协议事务。然而，对于有效块的处理器写会产生 BusWrite 请求，但不会改变状态。总线一侧的请求也可能会导致状态转变。比如，对于一个有效块，如果收到一个和该有效块匹配的 BusWrite 和 BusRdX 请求，那么该数据块将会被无效掉，从而将导致一个进入无效状态的转换。

表 5-1 cache 协议的输入和输出

缩写	描述
PrRd	处理器读
PrWr	处理器写
BusRd	对块的读请求
BusWrite	写一个字到内存，并且无效掉其他拷贝
BusUpgr	无效掉其他拷贝
BusUpdate	更新其他拷贝
BusRdX	读一个块，同时无效掉其他拷贝
Flush	给请求 cache 提供一个块
S	共享线有效
\bar{S}	共享线无效

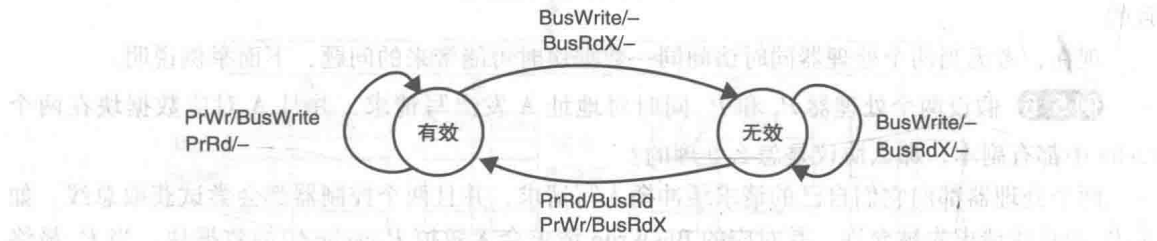


图 5-9 简单侦听 cache 协议的状态转换图

图 5-9 的状态转换图只给出了协议的高层行为，一个详细实现 cache 协议的状态转换图会包含更多的状态。在接下来的部分，状态转换图表示的都只是不同 cache 协议的高层行为而不是其具体实现。

5.4.3 侦听 cache 协议的设计空间

知道了基本的硬件结构，以及简单侦听 cache 协议的行为，现在我们考虑商用机器的 cache 协议设计空间。通过运行实际程序可以发现简单协议中存在的很多性能限制，而这些 cache 协议就是要解决这些问题。

尽管简单，但是之前介绍的简单协议有很严重的潜在性能瓶颈，一个明显低效的地方就是所有的写请求都会发送总线事务。实际上，不管多处理器是运行多个独立的程序还是一个并程序，处理期间共享的内存块都是很少的，绝大部分块是单个处理器独占访问的。因此，如果能将访问这些非共享的读和写请求在本地 cache 解决，而不发送需要同其他 cache 交互的总线事务，那就明显获益。接下来，假设 cache 是写回的。对于非共享块的访问不应该影响其他 cache，这是我们下一个要讨论的具体协议的基础。

回忆一下，在支持写回的 cache 中每个数据块有两个状态位：一个有效位（V）和一个修改位（M），后者有时也称为脏位。现在，我们给出支持写回的 cache 协议的高层行为，我们称

这种 cache 协议为 MSI 协议。图 5-10 给出了对应的状态转换图。之所以称之为 MSI 协议，是因为它有 3 个状态：修改状态（M），共享状态（S），无效状态（I）。

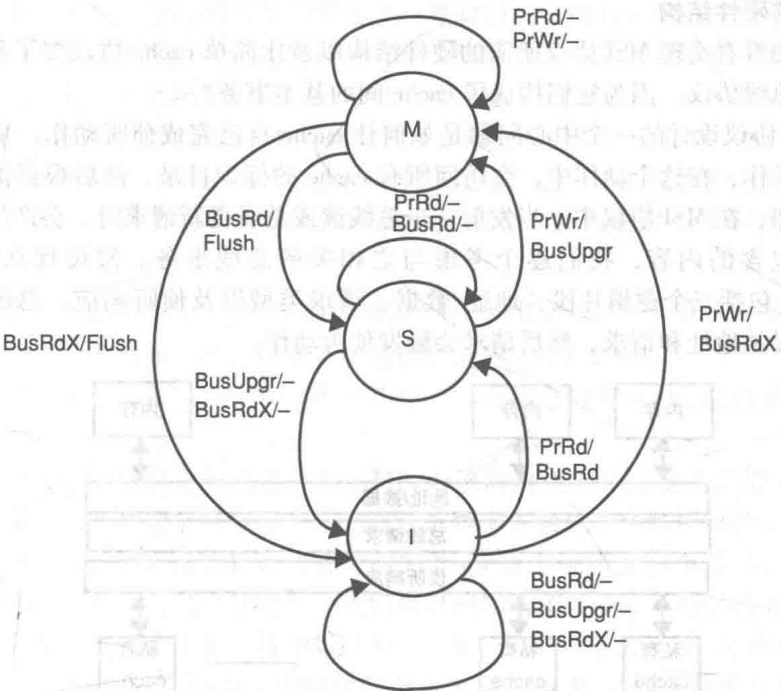


图 5-10 MSI 协议的状态转换图

MSI 协议的行为特征

下面说明图 5-10 中 MSI 协议的三个状态。修改状态（M）表示本地 cache 的备份块是系统中唯一保持最新数据块。因此，处理器的读和写请求可以完全在本地完成，而不需要同其他 cache 交互。由于 cache 采取写回策略，因此不必将更新的块写回，这也导致内存中的块保存的是旧值。另一方面，处于共享状态（S）的 cache 块除了这个本地块外，在内存和其他远程 cache 中可能还有多个备份，此时内存中的数据是最新的，或者说是干净的。对于共享状态的 cache 块的读请求和远程 cache 无关，可以在本地完成，但是，对于写请求，必须无效掉远程 cache 中的相应块。最后，无效状态（I）同简单协议中的无效状态相同，即本地 cache 的备份块是无效的，或者本地 cache 中没有该块。因此，处理器的读写请求都需要查询其他 cache 和内存才能完成。

我们仔细观察一下与处理器读写请求相关的 cache 一致性事务与简单协议相比有什么不同。我们先假设内存数据唯一有效的备份就是内存中的数据块。处理器读取数据块会触发从 I 状态到 S 状态的事务。对于同一个处理器接下来对该块的写请求，cache 控制器会发射一个总线更新请求（bus-upgrade request, BusUpgr），该事务会无效掉所有远程 cache 对该块的备份，本地 cache 块的状态会从 S 转到 M。此后，该处理器的后续读写请求就可以在本地完成了。

考虑另外一种情况，假设处理器 P_i 和 P_j 的 cache 中都有内存块的有效备份，两个备份的状态都处于 S 状态。如果 P_i 对块进行写，那么 P_j 的 cache 块必须被无效掉。这会导致 P_i 的 cache 中的块状态从 S 变为 M，这点与简单协议不同——在简单协议中不会有状态变化。假设接下来 P_j 再次读该块，与简单协议不同的是，在简单协议中，可以从内存中得到读失效的数据。在 MSI 中，唯一更新了的块在 P_i 的 cache 中，这就需要 P_i 的 cache 提供 P_j 所需要的块，这一点对于侦听 cache 协议的实现非常重要。注意， P_i 向 P_j 的 cache 提供块的同时也会更新内

存，这样内存可以保存对共享状态块的最新数据。在状态转换图中，这个操作被标注为刷新，它会出现从 M 态转换到 S 态，以及从 M 态转换到 I 态的过程中。

MSI 协议的硬件结构

现在来仔细看看实现 MSI 协议所需的硬件结构以及比简单 cache 协议多了些什么。我们主要关注底层的总线协议，因为它们构成了 cache 间的基本事务。

侦听 cache 协议设计的一个中心问题是如何让 cache 自己完成侦听动作。每个总线事务都包含一个侦听动作，在这个动作中，会访问所有 cache 的标识目录，然后根据协议会采取某些具体措施。比如，在 MSI 协议中，当发射一个总线读或总线更新请求时，会产生侦听动作。为了理解这其中包含的内容，我们逐个考虑与之相关的总线事务。假设现在的总线模型如图 5-11a 所示，包括三个逻辑片段：地址/数据、请求类型以及侦听响应。总的来说，总线事务会先给总线提供地址和请求，然后请求会触发侦听动作。

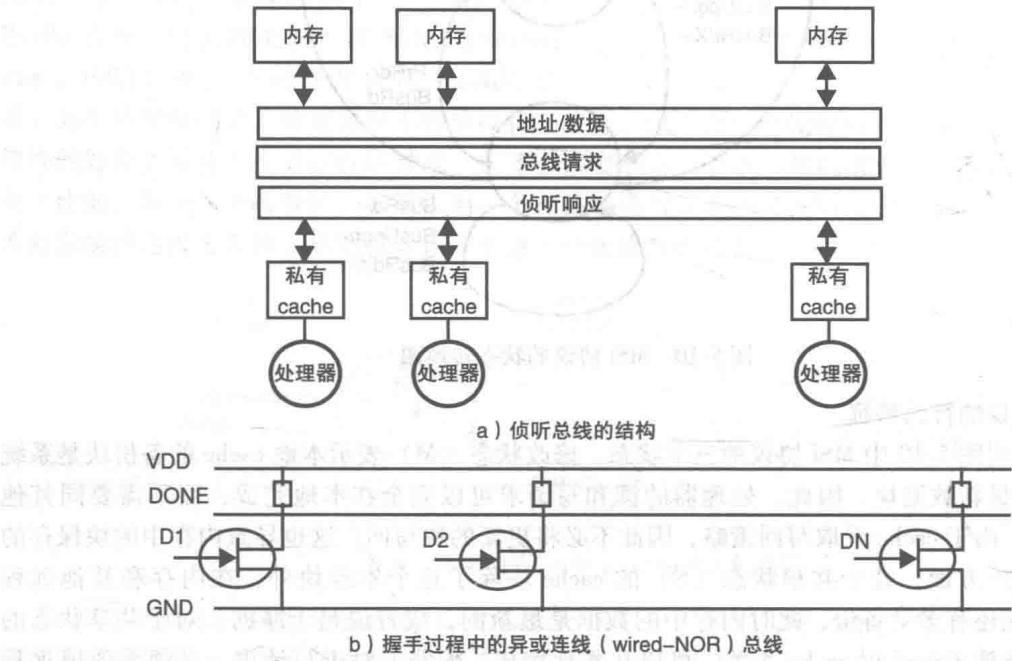


图 5-11 侦听总线

第一个问题是，要使所有的 cache 实施侦听动作，地址和请求需要在总线上保持多久？我们希望这个时间越短越好，因为这直接影响 cache 失效的开销。这个时间由多个方面决定：总线长度、连接的 cache 数目以及检查每个 cache 标识的所需时间。在每个处理器节点，来自处理器一侧和总线一侧的标识查找操作可能存在冲突，这种冲突导致标识检查的时间发生变化。

有两种基本方法可以实现侦听动作：异步侦听 (asynchronous snooping) 和同步侦听 (synchronous snooping)。在异步侦听中，当所有远程 cache 完成侦听动作后，需要给本地 cache 一个握手信号。握手信号的一个典型实现方法如图 5-11b 所示。这种方法计算 D1 到 DN 的异或值。其实现是通过被动地将每个晶体管通过一个电阻连在 VDD 上。如果输入信号被确定为逻辑 1，那么对应的晶体管会对地产生 DONE 输出信号 (GND 或逻辑 0)。因此，要使 DONE 变为逻辑 1，所有的输入都要求是 0。先对每个 D 输出恒定的逻辑 1，当执行完标识检查后会输出逻辑 0，通过这种方法，可以在所有远程 cache 都完成标识检查时，通知本地 cache。当所有操作完成后，DONE 被置为 1。

所有异步协议都存在一个缺点，那就是握手需要花费额外的时间。在我们讨论的情况中，

这个额外时间包括：所有 cache 控制器首先通过驱动 DONE 信号变为逻辑 0 来对当前地址进行响应，然后通过侦听行为来报告该动作已经完成。异步协议的替代方法之一是同步侦听协议，在同步协议中，需要确立一个时间上界，在这个上界时间内，所有 cache 都能完成标识检查并且进行响应。这个上界还要考虑来自处理器请求和总线请求的冲突。一种使最坏情况时间接近平均时间的方法是，复制多个标识目录，一个对应处理器一侧的请求，另一个对应总线一侧的请求。总线一侧的标识目录经常被称为对偶标识目录。需注意的是即使复制了标识目录，还需要保持两个目录的更新以确保一致性。在所有 cache 完成之前，这一开销也要被计入最坏情况时间。

另一个影响侦听设计的方面是从侦听操作返回的信息。我们通过下面的例子来说明这个问题。

例 5.7 假设一个处理器发出了一个读请求，但在本地 cache 中失效了。那么，什么情况下应该由内存响应总线读请求？什么情况下又应该由远程 cache 来响应？

如果该数据块在某个远程 cache 中的状态是 M，那么这个远程 cache 应该响应。而在其他情况下，应该由内存响应。

这个例子提出了一个很有趣的问题，为了判断应该由内存响应还是由远程 cache 响应，需要增加哪些硬件支持？同其他总线事务一样，总线远程事务会先产生侦听动作，就像之前描述的那样。一旦侦听完成，拥有最新修改过备份的那个 cache 必须产生一个信号，表明它拥有唯一正确的备份，内存中的备份已经过时了。我们可以利用同图 5-11b 类似的实现方法，但是重新定义输入和输出来完成这个工作。将 M1 到 MN（而不是 D1 到 DN）作为输入，将 REMOTE（而不是 DONE）作为输出。当 cache *i* 中的块状态为 M 时，将 *M_i* 置位。如果有某个 cache 中的块为 M 状态，REMOTE 就被清零，表示那个远程 cache 将会响应。在总线事务的下一阶段，远程 cache 会将块送给需要的 cache，同时，内存的块也会同步进行更新。

这种模式存在一个性能上的问题，当内存必须进行响应时，远程 cache 执行侦听动作所需的时间处在访存通路的关键路径上。但是，启动内存访问和执行侦听动作是可能并行执行的。只要还有某个 cache 块的状态是 M，那么内存就不能响应，这一点很重要。只有确保这些，内存才能完成总线上的传输。

除了读写请求要区别对待以外，简单协议和 MSI 协议的一个重要不同是，简单协议采用写穿透 cache，而 MSI 采用写回 cache。在写回 cache 中，当一个读请求失效后，如果这个失效导致一个 M 状态的块被替换出，那么这个备份必须要写回内存。这个被换出的块需要尽快写回，因为这个过程处在解决失效的读请求访存关键路径上。写回块时，为了避免一直等待总线，一个常用的办法是增加一个写回（或 victim）缓冲。victim 块一开始暂存在本地的 victim 缓冲中，这样读失效请求就可以尽早送到总线上。这里需要注意一个很关键的正确性问题——此时可能某个其他的 cache 发送读请求要访问 victim 缓冲中的块。对于这种读请求的正确响应是刷掉 victim 块，因为 victim 缓冲是整个系统中唯一有效的块备份。这样做的结果就是，写回缓冲也要参与所有的侦听动作（包括标识检查），就像 cache 一样。

进一步推广的 cache 一致性协议

现在介绍一种推广的（泛化类）cache 一致性协议，这个协议的目的是为了确定基本 MSI 协议的性能瓶颈。下面探讨 cache 一致性协议对于一组共享内存访问的性能影响。

在表 5-2 给出的例子中，处理器 1 先发射了一个对 A 的读请求，紧接着发送两个对 A 的写

表 5-2 包含 3 个处理器的访问序列实例

处理器 1	处理器 2	处理器 3
R_A		
W_A		
W_A		
	R_A	
		R_A

请求。第一个读请求，将内存中的块读取到处理器 1 的 cache 中，状态为 S。接下来的写请求会触发总线更新请求，并且在本地 cache 中，将块的状态改为 M。由于其他处理器中没有数据块 A 的备份，因此总线更新事务是没有必要的。如果新增加一个独占 (Exclusive, E) 状态，表明现在的这个 cache 块是系统中所有 cache 中唯一的备份，那么就on能避免上述的多余总线事务。我们稍后会讲解，如何将图 5-10 中的 MSI 协议状态转换图扩展成包括状态 E 的转换图。

回到我们的例子，由于块的状态为 M，这两个写操作执行时可以不产生总线事务。对该块的下一次访问是处理器 2 的读请求。这个读请求会 miss，然后会发送总线读请求，并由处理器给出响应。处理器 1 中的数据块状态会从 M 变为 S。接下来，处理器 3 同样发送对块的读请求，在本地同样会 miss，然后再次发送总线读请求。这个事务由内存解决，而不是处理器 1 解决，因为内存中的块已经是共享的并且更新过的。

处理器 2 和 3 的读失效有个基本的不同。第一个失效必须被一个远程 cache 响应，而第二个失效既可以由远程 cache，也可以由内存来响应。我们称第一种失效为脏失效，而第二种失效为净失效。在 MSI 协议中，净失效由内存响应。然而，如果 cache 到 cache 的传输比内存到 cache 的传输要快，那么也可以通过远程 cache 更快地解决。

IEEE 标准中的 MOESI 协议通过增加两个状态 E 和 O 解决了之前提到的性能问题。图 5-12 给出了 MOESI 协议的状态转换图，它解决了 MSI 协议的两个性能短板。第一个扩展是新增的状态 E (独占)。处在状态 E 的块备份是只读的，但是同 S 状态相反，它保证整个系统中，这是唯一的 cache 拷贝。这个特性消除了本地处理器读取该块时发送的总线读请求。为了实现这个多出来的状态，需要增加一个被称为共享的新握手信号。其硬件实现同图 5-11b 中的硬件结构类似，当 cache 需要某个数据块备份时，发送总线读请求时，侦听动作的结果表明这个块在 cache 中的状态是 E 还是 S。如果共享线是低电平，在图 5-12 中用 \bar{S} 表示，那么这个块就处在状态 E。否则，它处在状态 S。

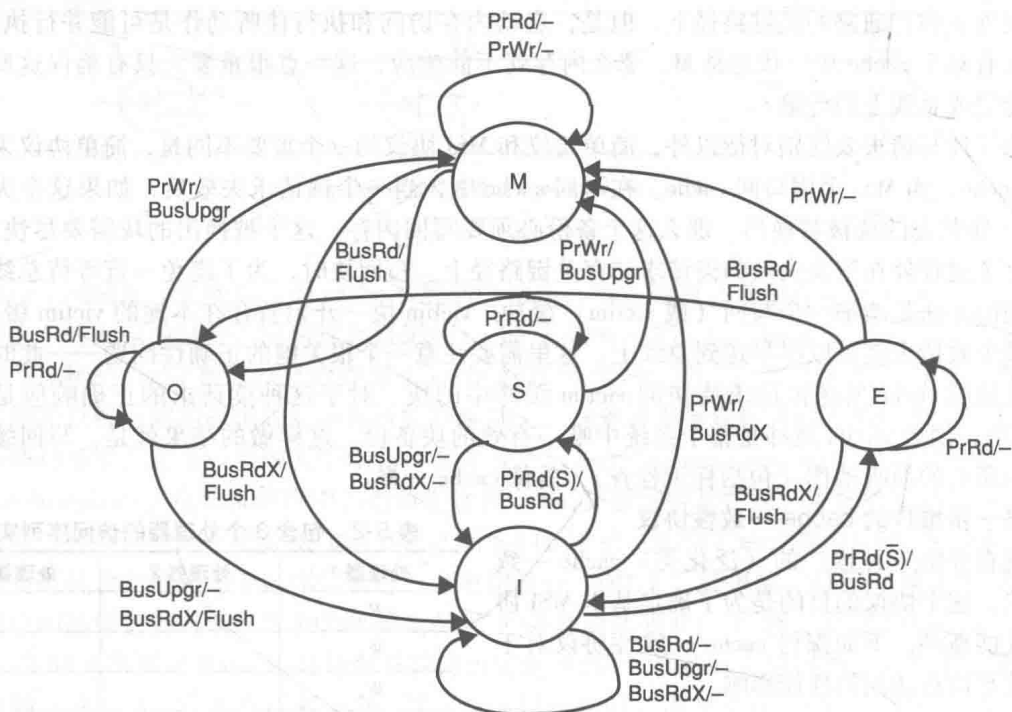


图 5-12 MOESI 协议的状态转换图

当使用远程 cache 的有效内存备份块来满足净失效的需求时,如果有多个有效的 cache 备份时,必须选择一个来提供数据。这一问题的常见解决办法是引入拥有权的定义。当 cache 拥有某个数据块时,它必须负责:当其他 cache 请求该数据块时,提供数据块备份;当块从 cache 中替换掉时,将拥有权归还给内存。如果拥有权被传送给内存,那么内存的数据也是更新了的。

对于状态为 E 或 M 的块,表示该块所在的 cache 就是数据块的拥有者。当存在多个 cache 备份时,则可以显式指定某个 cache 作为拥有者。图 5-12 的 MOESI 协议增加了一个新状态 O,即拥有状态,来跟踪块的拥有权。当某个 cache 包含一个状态为 E 的块,并且其他 cache 请求该块时,本地 cache 备份会变为状态 O,并且在将拥有权传给内存或其他 cache 之前,会对总线上任何一个需要该块的总线读请求进行响应。如果处在状态 O 的块被替换掉,那么这个块要被写回内存,并且拥有权也返回内存。如果处在状态 M 的块被替换掉,则拥有权被传给内存。最后,当一个处在状态 O、E 或 M 的块被无效掉时,拥有权会传给对这个块进行写的节点。

作为 MOESI 协议的一个子集, MESI 是现在商用机中极其常用的协议。出现这种情况的主要原因是,状态 E 的引入对于多道程序的运行环境来说,消除了大多数的总线更新请求,使得总线传递变少。

5.4.4 协议变种

在一些并程序中,有很多相同的读写共享模式,为此,我们现在考虑一些对于 cache 协议的优化。第一组优化是针对基于无效策略的 cache 一致性协议,例如 MSI 和 MOESI 协议。第二组优化针对的是基于更新策略的 cache 一致性协议。

基于无效策略的 cache 协议优化

生产者-消费者共享 (producer-consumer sharing) 是指有一个或多个生产者线程来修改数据,并有一个或多个消费者线程在数据修改后进行读取的程序行为。

例 5.8 考虑如下的读 (R_i)、写 (W_i) 操作序列,其中 i 代表处理器 i 执行的操作:

$$W_1, R_2, R_3, R_4, W_1, R_2, R_3, R_4, \dots$$

假设在操作序列开始前,所有的 cache 中都有备份块。那么在 MSI 协议下,这种访问序列会产生哪些总线事务呢?

由于所有 cache 最初都有块的备份,处理器 P_1 的写操作会发送总线更新请求,使得其他 cache 中的备份都无效。接下来,这些处理器发送读请求,导致 3 个总线读事务。此后,这种访问模式再多次重复进行。最终的结果就是,这个序列中的所有处理器访问操作都会产生总线事务。

在每个写操作后的三个总线读事务都会导致同一个块在总线上传输,这不仅浪费了宝贵的带宽,此外,由于将数据块装载到 cache 中有一定的延迟,因此也会影响那些要发起读操作的处理器。有一种叫作读抄写或读广播的方法,可以在 P_2 的 cache 控制器发出块的读请求时就将块装载到所有的三个 cache 中。当读取的数据块返回时,所有保存该块备份但处在无效状态的 cache 就会从总线上直接获取到这个块。这样做的一个缺点是,块可能会被装载到一个并不会访问它的 cache 中,从而造成 cache 污染。另外,由于可能会与处理器访问冲突,将块装载到 cache 中也有一定的额外开销。

另一种优化方案解决的是迁移共享问题。在绝大多数的并程序中,共享数据的修改发生在临界区中。考虑图 5-2 所示的并程序临界区:

```
LOCK(LV);
    sum += mysum;
UNLOCK(LV);
```

临界区的语义保证对 `sum` 的读、修改以及写回是原子的。由于对 `sum` 的读-修改-写带来了先读后写的操作, 假设三个处理器 P_1 , P_2 和 P_3 连续地进入临界区, 在 MSI 协议中会连续发生下面的总线事务序列:

`BusRd1, BusUpgr1, BusRd2, BusUpgr2, BusRd3, BusUpgr3, ...`

这种访问模式就是迁移共享。在上面的序列中, `cache` 控制器发射一个总线读, 接着发射一个总线更新, 这两个事务都由最近拥有块且状态为 `M` 的 `cache` 响应。一种显而易见的优化方法是, 当一个块拷贝请求发生时, 用一个 `BusRdX` 请求来取代分开的 `BusRd` 和 `BusUpgr`。采用这种优化后, 总线事务序列就会变成下面的样子:

`BusRdX1, BusRdX2, BusRdX3, ...`

这就减少了每次对迁移块读-修改-写时需要的 `BusUpgr` 请求。为了利用这种优化, 协议需要有一种方法来发现迁移共享、启用优化以及关闭优化。有一种被证明可以鲁棒工作的方法: 当 `cache` 发射总线读请求时, 如果恰好只有一个 `cache` 拥有这个块的有效备份, 那么就启用优化。例如, 当 `BusRd2` 发射到总线上时, 唯一块备份在 P_1 的 `cache` 中。 P_1 通过查看块的状态为 `M`, 可以知道它拥有唯一的一个备份, 就会将总线读请求转化为总线独占读请求, 并且传输块的拷贝。为了进一步观察一个块在什么时候结束迁移共享, 考虑下面的例子:

`BusRd1, BusUpgr1, BusRd2, BusUpgr2, BusRd3, BusRd4, BusRd5, ...`

这个序列和之前的主要不同在于, 这个块的迁移共享一直持续到 P_3 发送对块的拷贝请求。然而, 下一个对块的访问是 P_4 , 而不是 P_3 对块的修改。这时如果不关闭优化就会出现问題, P_4 将会得到对块的一份独占的拷贝, P_3 接下来对该块的访问就会失效。这种失效在 MSI 协议下是不应该发生的。幸运的是, 如果当一个处理器对块的读请求后面紧跟着另一个处理器的读请求, 并且后面的读请求发生在前一个处理器修改块之前, 那么此时就可以关闭优化。

通过对迁移共享的优化来扩展 MSI 协议非常直接。在探测过程中, 状态 `S` 被分为两个状态: `S2` 和 `S`, `S2` 指恰好有两个拷贝, 而 `S` 指有一个或多于两个拷贝。通过下面的过程可以发现迁移共享。当 `cache i` 中有一个处于状态 `M` 的块, 并且 `cache j` 发射了总线读请求, 则 `cache i` 中的块状态改为 `S2`, 将块送入 `cache j` 并且状态设为 `S`。在处理器 j 发射下一个写请求时, `cache j` 发射一个更新请求, 此后该块被认为是迁移的。为了跟踪迁移块, 协议增加了两个状态: 干净迁移和脏迁移。干净迁移是指块在优化开启的时候被装载的, 但是还没有在本地被修改。脏迁移是指在优化开启时装载的块, 并且已经被修改过了。当对干净迁移块有一个来自其他 `cache` 的读请求时, 对该块的迁移优化就会被关闭。这样做的原因是, 在迁移共享的访问模式中, 处理器发射的 `load` 指令和紧接着的 `store` 存储块的指令之间, 是不会插入有其他处理器的 `load` 操作的。

基于更新策略的 `cache` 协议优化

目前为止, 我们讨论的维护 `cache` 一致性的协议都是尽早清除旧的拷贝。这对于数据共享较少的多进程工作很有效, 但是当共享量很多时就没那么有效了, 例如, 在生产者-消费者共享的情况下。即使有之前说过的读抄写优化, 基于无效策略的协议仍会导致所谓的一致性失效, 这种失效是由无效策略引起的。另外一种完全不同的方法是, 当某个块被修改时, 尽早更新所有远程 `cache` 中的备份, 而不是无效掉它们。

接下来介绍 20 世纪 80 年代用于 Xerox Palo Alto Research Center (PARC) 的 Dragon 多处理

中的基于更新策略协议的行为。这种协议的状态同 MOESI 协议相同，图 5-13 给出了它的状态转换图。图中为了简化省略了无效状态。同之前介绍的协议相同，状态的输入和输出由表 5-1 给出。注意这个协议也依赖于侦听远程 cache 拷贝的共享线。由于大部分与 MSI 协议相同，在这里不对所有的事务进行讲解，主要关注两种事件：读失效和写命中。

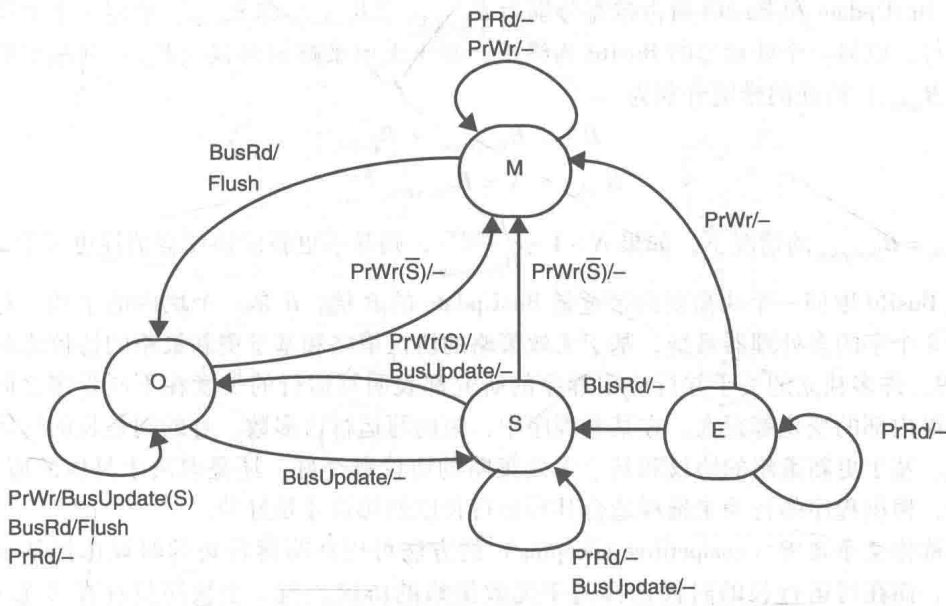


图 5-13 基于更新策略的 cache 协议的状态转换图（Dragon 系统）

读失效。同 MOESI 协议一样，如果某个处理器发生读失效，并且其他 cache 没有备份，那么就从中内存中装载数据块，并且状态置为 E。如果其他 cache 有备份，那么数据块的状态置为 S。

写命中。这种情况下，根据是否其他块有该块的备份，采取相应的动作。如果没有其他备份，即块状态为 E，则 cache 块的状态改为 M 并且不会产生总线事务，这与 MESI 相同。这种协议与基于无效策略协议的主要不同在于，如果此时其他 cache 仍有块备份，那么该采取什么动作。这个协议并不会发射 BusUpgr 来无效掉其他块，相反，它会发射一个总线更新（BusUpdate）请求来更新其他拷贝。更新是对于块状态为 S 或 O 的块。当一个块的状态为 O 或 S 时，其他 cache 中可能存在备份，因此，必须对共享线进行侦听。如果共享线未激活，表明没有远程拷贝块，将块状态改为 M 即可。如果某个处于状态 O 的块被替代了，拥有权会转移到内存或者其他含有状态为 S 的块的 cache，就像图 5-12 中的 MOESI 协议一样。为了简化，图 5-13 没有显式地画出无效状态，但是换出的数据块状态会隐式地改为无效状态。

一个重要的问题是，在哪种程序行为下，基于无效策略的协议或者基于更新策略的协议会更有效？在权衡二者时，一个非常有用的模型是写运行模型。

定义 5.2（写运行） 给定一个处理器读写序列，处理器 i 的读（写）被记为 $R_i(W_i)$ 。一个写运行是指同一个处理器的一连串写操作，直到出现其他处理器的读写操作。写运行的长度是指这一连串写操作的个数。

例 5.9 给定下列处理器对于某个块的读写序列，该序列中的写运行是哪些？长度为多少？

$$R_1, W_1, R_1, W_1, W_2, R_2, R_3, W_3, R_3, W_3, R_3, W_4, R_4$$

第一个写运行从处理器 1 的第一个写请求开始，到第二个处理器的写请求为止。这个序列

中有两个写操作，故其长度为 2。第二个写运行包括处理器 2 的单个的写，在处理器 2 的读请求后，处理器 3 的读请求是这个序列的终止。第三个写运行包括两个处理器 3 的写，到处理器 4 的写请求出现为止。

写运行的平均长度可以用来评价基于无效策略和基于更新策略的协议所消耗的带宽。假设 BusUpgr、BusUpdate 和 BusRd 所占带宽分别为 B_{BusUpgr} 、 $B_{\text{BusUpdate}}$ 和 B_{BusRd} 。给定一个平均长度为 N 的写运行，以另一个处理器的 BusRd 为终止，基于无效策略的协议 (B_{inv}) 和基于更新策略的协议 (B_{update}) 消耗的带宽分别为

$$B_{\text{inv}} = B_{\text{BusUpgr}} + B_{\text{BusRd}}$$

$$B_{\text{update}} = N \times B_{\text{BusUpdate}}$$

在 $B_{\text{BusUpgr}} = B_{\text{BusUpdate}}$ 的情况下，如果 $N > 1 + \frac{B_{\text{BusRd}}}{B_{\text{BusUpdate}}}$ ，则基于更新的协议会消耗更多带宽。

假设 BusRd 取回一个块需要的带宽是 BusUpdate 的 B 倍， B 是一个块内的字数。对于一个块大小为 8 个字的多处理器系统，基于无效策略的协议策略和基于更新策略的协议之间的平衡点是 $N=9$ 。许多独立的关于并行应用程序的研究都表明写运行的长度在不同程序之间，甚至在单个程序内部的变化都很大。在某些程序中，短的写运行占多数，有的则是长的写运行占多数。因此，基于更新策略的协议和基于无效策略的协议哪个好，还是取决于具体的应用程序。这也表明，根据程序的行为来选择适合其写运行长度的协议才是好的。

一种称作竞争侦听 (competitive snooping) 的方法可以在写运行短的时候选择基于更新策略的协议，而在写运行长的时候选择基于无效策略的协议。当一个块拷贝存在多个 cache 中时，协议默认采用基于更新策略的协议。当某一个写运行的长度超过阈值时，协议会转变成基于无效策略的协议。我们将这个阈值记作写运行长度 (Write-Run Length, WRL)。为了触发向基于无效策略协议的转换，每个 cache 行都有一个辅助的计数器。在把一个块装载到 cache 中时，该计数器会被预设定为 WRL。每次对这个块产生 BusUpdate 时，会将计数器减 1，每次本地处理器访问该块时，将计数器设为 WRL。当计数器的值变为 0 时，cache 控制器会使本地的备份无效。当所有的块拷贝的计数器都变为 0 时，就没有哪个 cache 控制器可以激活共享线，如图 5-13 所示，块拷贝的状态变为 M。当下一次块的状态变为 S 时，协议会将该块切换回基于更新策略的模式。

5.4.5 多阶段侦听 cache 协议的设计问题

在图 5-7c 的机器模型中，每个处理器都有一级私有 cache，并通过总线互连，以便快速传递各种请求 (BusRd、BusRdX，以及 BusUpgr) 和响应 (侦听结果以及数据块传输)。然而在实际系统中，每个处理器可能有分层的多级私有 cache。这种本地的 cache 层级会引发侦听 cache 协议设计的一些新问题。此外，假定请求和响应在总线上能够即刻 (原子地) 传输也是不现实的，因为总线会在整个事务过程当中被占用。例如，一个总线读请求包括所有 cache 侦听响应的的时间，下式给出了总线被占用的时间：

$$T_{\text{transaction}} = T_{\text{arbitrate}} + T_{\text{request}} + T_{\text{response}}$$

在整个事务过程中，一直保持总线被占用是我们不愿意看到的，因为这会大幅降低总线的可用带宽。我们希望将请求和响应进行分离，这样的话，当在等待内存或其他 cache 将数据块传到总线上的过程中，其他事务仍然可以利用总线。在本节中，我们将探讨分离事务总线 (split-transaction bus) 以及多级私有 cache 对于侦听 cache 协议设计的影响。图 5-14 给出了对应的机器模型。

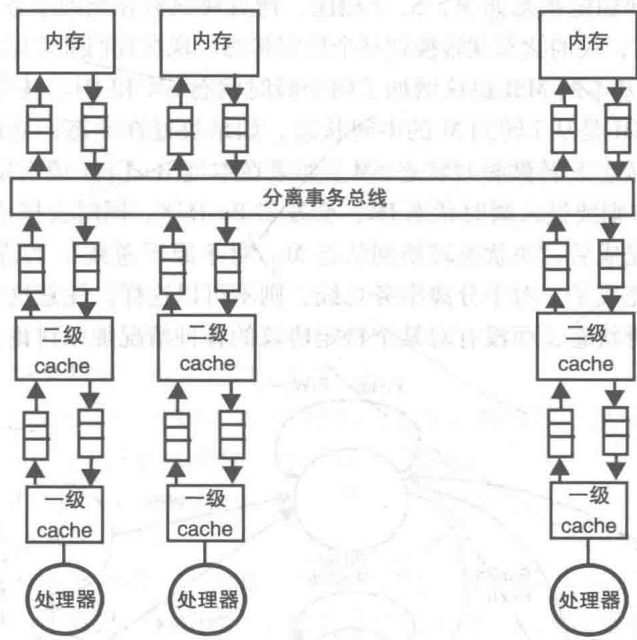


图 5-14 拥有多级私有 cache 以及分离事务总线的多处理器模型

在分离事务总线中，一个连续的事务被分成一系列子（分离的）事务。一种很自然的划分是将其分为请求阶段和响应阶段。由于一个连续的事务被分成几个阶段，我们把本节的 cache 协议称作多阶段侦听 cache 协议。由于不同的 cache 和内存模块不会以同样的速率进行请求，我们在图 5-14 中模型的单元之间加入了 FIFO 的请求缓冲，这样可以消除单元间的速度差异。由于内存请求总体上并不会在所有内存模块之间均匀分布，因此在不同节点上，同一个内存模块所占据的 FIFO 缓冲项个数可能会不同。同样，由于现代处理器允许同时有多个处理中的内存请求，因此不同 cache 的入队和出队所占据的 FIFO 缓冲项可能也不尽相同。在本节，我们只给出了多阶段侦听 cache 协议的大体行为，在第 7 章，我们将再建立一套框架，用于推断 cache 一致性和存储一致性的正确性。

瞬时（非原子的）cache 状态

目前为止，在我们讨论过的 cache 协议的状态转换图中，cache 控制器可以在发射请求时就决定下一个状态是什么。当一个连续的事务被分离成多个阶段以后，这种做法就不再成立了。即使是比图 5-14 中的模型更简单的、只有一级私有 cache 和原子总线的模型，也必须等到请求被发射到总线上、并且得到侦听结果后才能决定转向哪个状态。以图 5-13 的 Dragon 协议为例，考虑当处理器写一个状态为 0 的拷贝块的动作。下一状态取决于共享线的状态，并且在得到侦听结果后才能做出决定。图 5-12 给出的 MOESI 协议也是类似，处理器对一个状态为 I 的拷贝块进行读操作时的动作，需要根据共享线的状态才能决定下一状态是 E 还是 S。

由于一个连续事务不再是原子的，本地备份块的状态可能在事务发送的过程中发生变化。例如，考虑图 5-10 的 MSI 协议，假设当处理器发射一个对块的写请求时，块的状态为 S。再假设 BusUpgr 请求被发送到总线之前，一个远程 cache 对同一个块发射了 BusUpgr 请求。这个更新请求无效掉了本地备份。更重要的是，本地 cache 发射的 BusUpgr 将不再有用，因为此时它应该发射一个 BusRdX 请求。

为了解决非原子的多阶段事务的问题，我们需要瞬时（或非原子的）状态。之所以称它

们为瞬时的，是因为同稳定状态如 M，S，I 相比，拷贝块只有在当前事务未完成时才会处在瞬时状态。当事务完成后，块的状态就转换到某个稳定状态。现在我们来考虑如何处理瞬时状态。

在图 5-15 中，对基本的 MSI 协议增加了两个瞬时状态 SM 和 IM。其中，SM 是指从 S 态转到 M 态的中间状态，IM 是从 I 转到 M 的中间状态。如果块处在 S 态，处理器发出写请求，块的状态就会从稳定的状态 S 转到瞬时状态 SM。如果在本地 BusUpgr 请求发送之前，远程 cache 发送了 BusUpgr 请求，则块进入瞬时状态 IM，会发射 BusRdX，同时去掉请求队列中的 BusUpgr 请求。最后，当事务完成后，块状态转换到状态 M。对于原子总线，当请求发射到总线上时，就可以认为事务已经完成了。对于分离事务总线，则不可以这样。注意这个例子只是从方法上大体解释了增加的瞬时状态，而没有对某个特定协议的各种情况加以讨论。

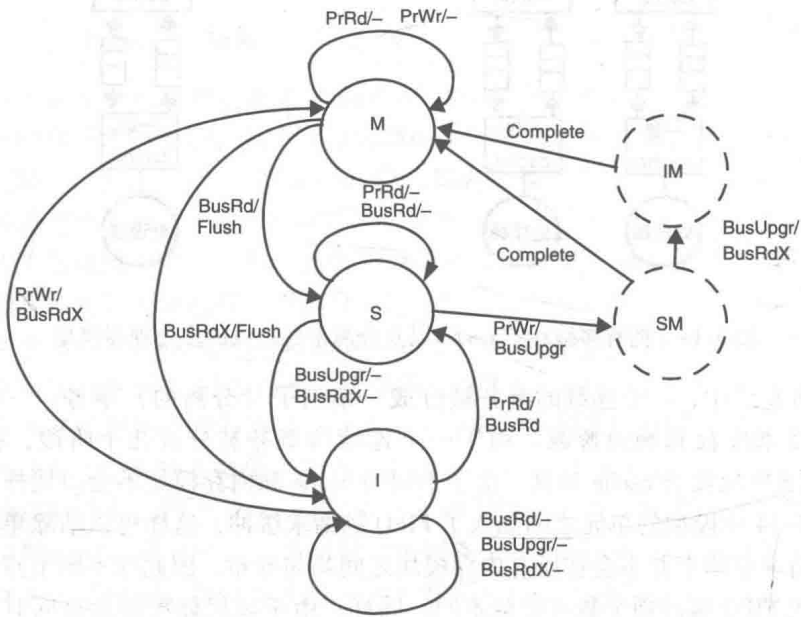


图 5-15 MSI 协议中的瞬时状态

分离事务总线上 cache 协议的设计问题

分离事务总线可以通过将一个事务分离成多个子事务来提高可用带宽。然而，不幸的是，由于对每个子事务都要有总线仲裁，这会增加延迟。因此，在设计一个分离事务总线协议的时候，需要在延迟和带宽之间进行权衡。

分离事务总线由于采用流水和总线事务重叠执行的方式，可以带来更高的带宽。一个活动可以流水化的前提是，它可以被分成多个更小的工作，并且这些小的工作可以按照一个严格的顺序执行。为了将流水化用在连续的事务中，需要按照图 5-16 将事务划分成请求阶段和响应阶段。

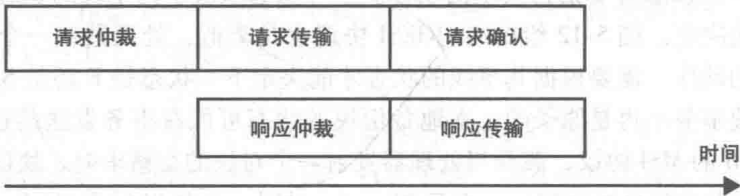


图 5-16 分离事务总线协议

在图 5-16 的协议中，请求和响应阶段按时间分开了。在 MSI 协议中，总线请求包括：BusRd、BusUpgr 和 BusRdX。响应包括：对于净失效从内存更新块，对于脏失效从其他 cache

更新块。请求信息中包括块地址和请求类型，响应信息中则包含了块内容、地址或者其他一些标识，这样更新的块可以同对应的请求匹配，并且存放到 cache 的正确位置上。由于请求和响应都要用到地址总线，不可以并行处理，除非有两条分开的地址总线（一条供请求使用，一条供响应使用），但这样的设计会带来很大开销。

在某些商业机器中采用的方法是，为每个请求分配唯一的标识符（例如请求号）。通过这个标识符将它同对应的响应联系起来。标识符的位长取决于总线上最多有多少未完成的事务，通常情况下，这个位数会比内存地址位数小得多。例如，对于 128 个未完成的总线请求，只需要 $\log_2 128 = 7$ 位标识符，而不需要 32 或者 64 位的地址。对于这种安排，请求要用到地址总线、请求类型总线以及标识符总线，而响应只用到数据总线以及一个独立的标识符总线。这样，请求和响应没有资源共享，也就可以并行执行了。

为了给分离事务总线设计一个健壮的 cache 协议，必须解决以下顶层问题：

- 如何处理冲突的请求（比如对同一个地址的请求）。
- 如何报告侦听结果。
- 如何防止缓冲区溢出。

当访问同一地址的请求冲突中，如果至少有一个是 BusUpgr 或 BusRdX，可以通过瞬时状态来解决。在原子总线的情况下，当请求被发射到总线上时，就可以检测到冲突。然而，在分离事务总线的情况下，并不总是这样，这是因为在任意时刻可能总线上有多个事务挂起，处理请求冲突非常复杂。

20 世纪 90 年代制造并应用的 SGI Challenge 机器提出了一种方法，就是在每个节点维护一张全系统的表，这个表记录了整个系统中未完成的请求。在发射请求之前，先去查询请求表，只有没有与之匹配的表项，才能发送请求。在同一时刻，对每个内存块最多只有一个请求。在 SGI Challenge 中，请求表还被用来限制未完成的请求个数。只有请求表中有空表项的时候，才能发送请求，并且会分配给它一个请求标识符，后续的响应消息用它来和之前的请求联系起来。

第二个问题是如何报告侦听结果。由于侦听结果是在同一时间给出的，入站缓冲（从总线到 cache）中会存在问题，因为所有 cache 报告结果所花费的时间，取决于在缓冲区中拥有表项最多的那个 cache。总线被占用过长的时间可能会完全抵消掉分离事务总线带来的好处，因此侦听结果必须尽早报告。解决方法之一是，在将请求插入入站缓冲区之前，就查看 cache，这种方法也被 SGI Challenge 机器所采用。这样一来，侦听结果的报告就成为请求的一部分子事务。接下来，根据需要，请求就可以在缓冲区之间传播，并且可能在一个分离的响应阶段将更新的数据块送到总线上。很明显，只有当同一时刻，对某个数据块最多只有一个事务在处理时，这种方法才有效。

第三个问题是，如何处理缓冲区溢出。FIFO 缓冲可能会装满，从而无法插入新的请求。因此，在请求阶段需要查询是否所有的 FIFO 缓冲都有能力接收新的请求。要实现这一点，在请求子事务中有一个确认阶段，用来确认所有相关的 FIFO 缓冲是否有接收请求的空间。如果没有，就会返回一个否定确认，重新尝试请求，同样，图 5-11b 中的那种硬件实现可以应用到总线确认中。

在多级 cache 层次中维护 cache 一致性

最后一个问题是多级私有 cache 带来的影响。正如图 5-14 那样，每一级新的 cache 都会引入一组入站/出站 FIFO 缓冲，这样做的结果会使响应请求的时间变长。

包含关系的 cache 设计可以在很大程度上缓解这一问题（尽管包含设计并不是必需的）。一级 cache 和二级 cache 之间的包含关系是指，如果一个数据块在一级 cache 中，那么它也同时会存在于二级 cache 中。如果没有维护包含关系这一特性，侦听的结果必须要等到查询完一级

cache 之后才能得出。因此，如果没有包含，侦听结果的得出就会被延后。此外，所有的输入请求都会同处理器请求竞争一级 cache。综上所述，为了能尽快得出侦听结果，我们希望维护多级私有 cache 之间的包含特性。

通常很难自动保证 cache 之间的包含关系，除了对于一级和二级 cache 中的一些有特殊限制的 cache set。不过，也有一种方法可以用来维护包含特性，当一个块从二级 cache 中被换出时，强制一级 cache 中的对应块也被换出。这种方法的缺点就是每次二级 cache 中的块被换出时，都要访问一级 cache，这样做可能会影响处理器。另一个问题与同一级 cache 写命中时采取的策略相关。如果是写回策略，则一级 cache 和二级 cache 中的内容就不一致。这表明如果一个块被修改过，而其他 cache 访问它时，必须由一级 cache 进行响应。对于级联的 FIFO 缓冲，对于脏失效的响应延迟将显著变大。通过维护 cache 间的包含关系，并且在一级 cache 中采用写直达策略，两级 cache 中的数据块可以保持一致性，并且二级 cache 可以响应所有的脏失效请求，从而显著降低响应延迟。

5.4.6 通信事件的分类

设计内存系统的主要目标是降低处理器读写的平均延迟以及消耗的带宽。为了对 cache 协议的延迟和带宽效率有更直观的感受，我们首先来看 MSI 协议。表 5-3 给出了一个例子，这个例子中，三台处理器对包含 A、B、C 的块 B1 以及包含 D 的块 B2 进行访问。表中给出了每台处理器在每个时间片中的访问情况。前 3 次访问都会产生读失效，并将块 B1 读到自己的 cache 中。这些 cache 失效都是冷失效，也就是说都是处理器第一次访问数据块。第三台处理器进行了第四次访问，这次访问 B2 并将包含 A、B、C 的块 B1 从 cache 中换出。B1 被换出可能是由于映射地址冲突或者 cache 容量限制。第三台处理器接下来对 B1

表 5-3 三台处理器对内存块 A、B、C、D 的访问序列示例

时间步	处理器 1	处理器 2	处理器 3
1	R_A		
2		R_B	
3			R_C
4			R_D (换出块 B1)
5	W_A		
6		R_A	
7	W_B		
8		R_A	
9			R_C

的访问可能由于地址映射冲突而产生冲突失效，也可能由于 cache 容量有限而产生容量失效。冲突和容量失效都被归到替换失效这一类中，如果 cache 容量无限，则可以避免替换失效。

接下来，在第 5 步，第一个处理器修改了 A，这会产生 BusUpgr 请求，并且使处理器 2 cache 中的 B1 块无效。处理器 2 接下来对 A 的访问产生了读失效。这是由于一致性带来的失效，称为一致性失效。一致性失效扩展了第 4 章介绍的 3C 失效模型（冷失效，容量失效，冲突失效），加入了第 4 个 C（一致性失效），构成了 4C cache 失效模型。

第二台处理器的一致性失效后，在第 7 步，处理器 1 紧接着对 B1 中的 B 又进行了修改，接下来，处理器 2 再次访问 A，又会带来新的一致性失效。然后第 3 个处理器访问同一块中的 C，并且失效。这种失效不能称为一致性失效，因为数据块在第 4 步就由于容量不足或地址冲突被换出。更确切地说，我们将它归为替换失效。

仔细观察序列中的两次一致性失效，可以发现一些有趣的东西。第 6 步出现的第一个一致性失效是由于第一个处理器修改的字（位于地址 A）同第二个处理器访问的是同一个字。不同的是，第 8 步中第二个一致性失效是由于第一个处理器修改的字（位于地址 B）同第二个处理器访问的字（位于地址 A）不同。第二个失效可以忽略，这不会引起程序错误。这种情况下之所以会产生失效，是因为字 A 和 B 位于同一个块中，而 cache 一致性维护的粒度就是块，这种

情况下的处理器之间并没有交换数据。我们称第一个失效为真共享失效，而第二个失效为伪共享失效。真共享失效会将程序正确执行所需的数据读入 cache，也就是说存在数据的通信。

考虑另一个例子（表 5-4）。这个例子并不是我们通过直觉就可以判断的，因为引起失效时的操作所访问的数据（B）不同于被修改过的数据（A）。但是这种情况也应该被归入真共享失效，因为这次失效会将 A 的值引入 cache，并且第二个处理器接下来就会使用 A。总的来说，如果一次失效将新的数据取到 cache 中，并且在该数据块驻留在 cache 内时被访问了，我们就将这次失效认为是真共享失效。

同冷失效和替换失效类似，真共享失效是为了让程序正确执行，将新的数据读取到 cache 中。如果忽略这个失效，处理器就会访问到旧值，这类失效叫作必要失效，这同伪共享失效不同。伪共享失效并不会将处理器访问的新值取到 cache。我们可以忽略伪共享失效，而不会影响程序的正确性。因此，我们将这些失效称为非必要失效。另一种非必要失效会出现在采取写分配策略的 cache 中。假设对字 A 的写操作会将 A 所在的整个数据块读入 cache，如果该块在 cache 中的整个生存时间内只有字 A 被访问，这个失效就被认为是非必要失效，因为处理器间并没有发生数据通信。

通过逐个检查所有内存请求的 trace 记录，就有可能计算出不同种类失效的个数。在图 5-17 中，通过流图给出了一种将失效分为 4 类（冷失效，替换失效，真共享失效，伪共享失效）的方法。

这种分类算法需要记录每个处理器对数据块的第一次访问，从而记录冷失效的个数。为了区别替换失效和一致性失效，程序还必须记录块换出的原因——被替换还是被无效。

在某些情况下，失效的分类是很难界定的，比如下面的例子。

例 5.10 考虑表 5-5 中对同一数据块的处理器读写序列，分别给出冷失效、替换失效、真共享失效和伪共享失效的个数。

很明显，处理器 1 的第一次读请求会导致冷失效，因为该块之前从未被处理器 1 访问过。对于处理器 2 和处理器 3 的后续访问，将它们归为冷失效还是一致性失效（真共享或伪共享失效）就不是那么明确了。作为处理器 2 和处理器 3 的第一次访问，这两个失效应该被归为冷失效。但是，另一方面，处理器 1 对块的写请求都会总线发送无效请求，而不管处理器 2 或

表 5-4 导致真共享失效的访问序列示例

时间步	处理器 1	处理器 2
1	W_A	
2		R_B
3		R_A

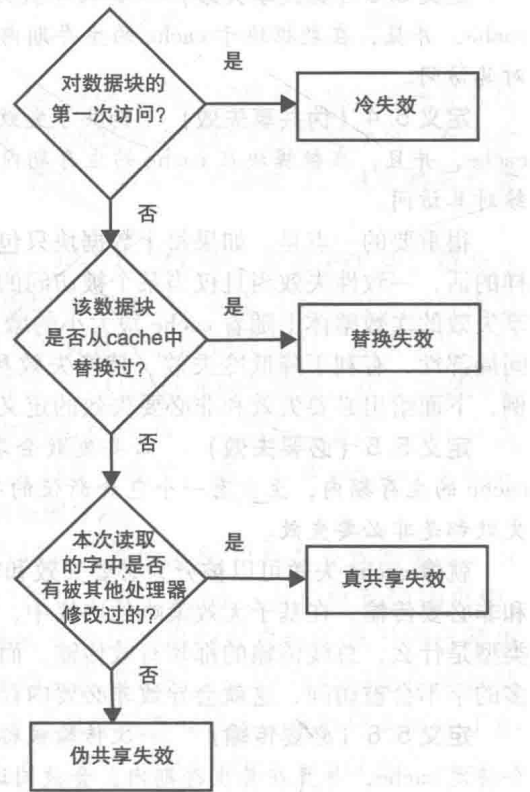


图 5-17 共享内存多处理器中 cache 失效的分类

表 5-5 失效分类示例

时间步	处理器 1	处理器 2	处理器 3
1	R_A		
2	W_A		
3		R_A	
4			R_B

处理器3是否访问过该块。在图5-17描述的分类算法中,冷失效优先于一致性失效,因此本例中的3次失效都被归为冷失效。

为了区别真共享失效和伪共享失效,在块驻留于cache期间,分类算法必须跟踪块中被访问的字。一个块处在cache中的时间被称为块在cache中的生存期(lifetime)。当出现以下3种事件之一时,生存期会被终结:替换;无效;程序终止。

下面给出真共享失效和伪共享失效的定义。

定义5.3 (真共享失效) 真共享失效会将之前由于被无效掉而换出的数据块重新取回cache,并且,在数据块于cache的生存期内,即使某个字被其他处理器修改了,仍然可以继续对其访问。

定义5.4 (伪共享失效) 伪共享失效也会将之前由于被无效掉而换出的数据块重新取回cache,并且,在数据块在cache的生存期内,如果某个字被其他处理器修改了,那么就不会继续对其访问。

很重要的一点是,如果每个数据块只包含一个字,那么伪共享失效就可以完全避免了。这样的话,一致性失效当且仅当某个被访问的字被其他处理器修改过时才会出现。实际上,伪共享失效的次数整体上随着cache块大小的增加而增多。另一方面,更大的块可以提供更好的空间局部性,有利于降低冷失效、替换失效和真共享失效。伪共享失效是非必要失效的一个实例,下面给出必要失效和非必要失效的定义。

定义5.5 (必要失效) 必要失效会取回一个数据块到cache中,并且在这个数据块在cache的生存期内,至少有一个包含新值的字被访问过,这类失效成为必要失效,而其他所有失效都是非必要失效。

就像cache失效可以被分为必要失效和非必要失效一样,内存传输也可以被分为必要传输和非必要传输。在基于无效策略的协议中,如果数据块大小是一个字,那么不管其地址和请求类型是什么,总线传输的都是有效传输。而随着块大小的增大,一个块在其生存期内,越来越多的字不会被访问,这就会导致非必要内存传输。

定义5.6 (必要传输) 一次传输被称为是必要的,当且仅当这次传输将一个新值送给某个特定cache,并且在其生存期内,会被同这个cache相连的处理器访问。

对于基于无效策略的协议,非必要传输只包含那些没有新值的字,比如,它们在块的生存期内都没有被访问。对于基于更新策略的协议,非必要传输包括所有通过更新操作向远程cache传播、但是又没有被远程处理器所访问的数据字。

5.4.7 TLB 一致性

cache一致性的作用是能够保证处理器在访问每个单独的内存位置时,都好像只有一个内存拷贝似的。作为软硬件接口暴露出来的存储模型确保了每个内存单元值的一致性,不过,光靠内存是无法覆盖并行程序正确执行所需的全部状态的。例如,对于寄存器中的变量值来说,解决cache一致性的方法无法保证所有的处理器都以一致的方式访问它们。因此,就需要依靠编译器或者应用程序开发人员,来保证寄存器中的变量只能被某个处理器的线程访问到。操作系统设计者必须保证,与系统正确提供服务相关的全部关键状态都是一致的,而虚拟存储管理就是操作系统提供的关键服务之一。

在虚拟存储系统中,整个虚地址空间被划分为固定大小的虚页。无论是串行程序还是并行程序,都要访问虚地址空间。某些被访问的页在物理内存中,但是其他页只在硬盘中有备份。访问不在内存中的页会引起缺页异常。在处理器真正能访问该页之前,操作系统的服务程序会将页调入物理内存。这就需要对每次内存访问都做检查,判断访问的页已经在内存中还是需要

从硬盘调度进来。对每一次访存，都要进行虚实地址转换，这是通过页表实现的，页表包含了每个页的虚实地址等信息。因此每次读写内存都需要查询页表，如果没有相关硬件的协助，采用虚拟内存带来的性能开销将非常大。

为了加速地址转换和虚拟内存访问的确认过程，引入了旁路转换缓冲（TLB）这一关键硬件。TLB 中缓存了最常被访问页的整个页表（PTE）。PTE 中包含了虚页到物理页的映射、访问权限、访问历史，以及是否在内存中被修改过等信息，下图给出了表项的内容：

虚页号	物理页号	页保护	引用位	脏位
-----	------	-----	-----	----

对于采用物理地址索引的 cache，TLB 位于处理器和一级 cache 之间。它将每个地址的最高几位——虚页号，转换到指向物理内存中的物理页号。为了帮助缺页处理程序选择替换页，有时会增加一些引用位，并且在每次访问该页时进行更新。此外，如果页被修改过，就会将脏位置位，从而保证页被换出时写回硬盘。

TLB 的作用相当于对页地址转换的 cache。如果某一项不在 TLB 中，而在 PTE 中，就会产生 TLB 故障（fault）或 TLB 失效，将失效的 PTE 表项取到 TLB 中。如果页表中没有对应虚页的物理地址，即发生了缺页（page fault），就会将缺失的页从硬盘取到内存中。如果发生访问权限冲突，也会产生 TLB 故障，例如，某个进程企图对一个只读页进行写操作。在多处理器系统中，每个处理器都有自己的 TLB，作为页地址转换的私有 cache。类似私有数据 cache 存在 cache 一致性的问题，TLB 也面临着一致性问题，称为 TLB 一致性问题（TLB consistency problem）。在本节，我们主要讨论商用机中对这个问题的处理办法。

当 PTE 在某个 TLB 中的备份同内存中的 PTE 不同时，就会出现 TLB 不一致的情况。首先，将虚页映射到物理页时，如果虚存管理将某页换出，并且将另一个页放到物理页中，此时就可能发生不一致。因为此时，旧的 TLB 会将刚从内存换出的虚页继续映射到已经更新内容的物理页上。此外该物理页中所有缓存的数据块也都过时了，因为它们保存的仍然是旧的虚页位置。为了维护一致性，TLB 中的所有过时项都需要被移出，此外所有缓存了被换出页的备份块也要被无效掉。其次，在某些情况下，多个 TLB 项中页访问权限位的不一致也会导致错误。比如，某个页的访问权限初始时对所有处理器都是可读可写的，后来，权限变为了只读，这一变化必须在 PTE 的所有 TLB 备份中都体现出来，否则有的处理器就可能违反权限。如果对某个页的访问权限从只读变为读写，并且这一变化并没有在所有的 TLB 中反映出来，那么这种权限的放宽就没有真正起到作用，并且会产生不必要的越权访问异常。再次，不同拷贝的访问历史信息（引用位）之间的不一致不会引起正确性问题，但在虚存管理选择换出页时可能会产生次优的选择。最后，被修改过的页必须正确记录，这也是为什么正确设置脏位非常重要。

要想保证操作行为的正确性，必须确保 TLB 中 PTE 的某些关键信息（例如地址转换、权限降低或脏位）的一致，而其他 PTE 信息（权限放宽或访问历史位）的不一致只会对性能产生影响，因此这类一致性的保证并不是必需的。

我们现在介绍一种解决 TLB 一致性问题的方法，这种方法中，TLB 是由软件处理程序管理，并且 cache 是按物理地址寻址的。我们通过下面的典型情景进行讲解：一个处理器对不在内存中的某页进行访问，缺页异常会触发虚存管理（软件）进行处理，然后某个处理器会执行虚存管理代码，选择当前在物理内存中的一个页进行换出，为了避免 TLB 中出现旧的 PET 项以及 cache 中出现旧的备份块，这一操作需要通知所有相关的 TLB 和 cache。幸运的是，由于 TLB 是通过软件管理，这样可以在内存中记录所有包含该转换的 TLB 项备份以及所有包含

该虚页的 cache 块。页面映射的改动必须通知到这些 TLB 项和 cache 块，这一通知过程通常由一个称为 TLB shutdown 的过程来执行。

在 TLB shutdown 中，执行虚存管理代码并且负责完成地址映射变换的处理器首先锁定 PTE，然后有针对性地所有相关处理器发送中断信号。当接收到中断信号时，每个处理器都会调用一个软件处理程序来将本 TLB 中的旧 PTE 项清除（通过一个简单的无效化操作），并且还会无效掉本地 cache 中属于这个物理页的所有数据块拷贝。和这次 shutdown 操作相关的所有处理器都会对发起 shutdown 操作的处理器进行应答。一旦所有旧的虚实映射关系被清除，TLB 之间就一致了，此后释放对 PTE 的锁定。由于对所有影响正确性的页表变换都需要执行类似的操作序列，因此这种方法很明显降低性能。尽管 TLB shutdown 操作在实际系统中很少出现，但是我们依然有必要通过适当的硬件支持来降低这一开销。

5.5 可扩展共享内存系统

对于总线系统来说，一个很大的局限就是它只能连接数量较少的 cache 和处理器。总线的带宽受限与连线数量与时钟频率的乘积。当节点数目增多时，总线带宽会降低，因为连线的长度以及传输量随着节点数目增多而增长。在本节中，我们考虑如何将共享内存的地址空间扩展到更多节点。理想情况下，随着节点数目增多，内存带宽也应按比例增加，并且内存延迟保持不变。

图 5-7 中的多处理器组织方式无法扩展到大规模节点，原因至少有两个：第一，总线无法适应大量节点。实际上，它只能适应很少数量的节点；第二，图 5-7 中的 dance-hall 组织方式（处理器在互连网络的一侧，存储模块在另一侧）有一个特点，就是任意处理器访问任意存储块的访问延迟几乎相同。然而，当节点数增多时，由于互连网络的延迟增加，所有的访存延迟都将增加。

相比来说，图 5-18 中的组织方式很明显具有更好的可扩展性。第一，它用通用互连网络替代了总线，这个网络所能提供的带宽可以随着节点增多而线性增长，并且访问延迟随节点增多呈亚线性增加。第二，内存分布到各个节点中，这样做的依据是访存局部性——数据通常主要由单个处理器进行访问，将这些数据放在离处理器近的存储中可以显著降低访问延迟。显然，当多个程序在多处理器上并发执行时，程序代码以及进程/线程的私有数据都有很好的局部性。这一结构对并行程序作了扩展，它对共享数据结构进行了划分以便更好地开发局部性，因此，将存储资源分布到不同的节点上也就是顺理成章的了。

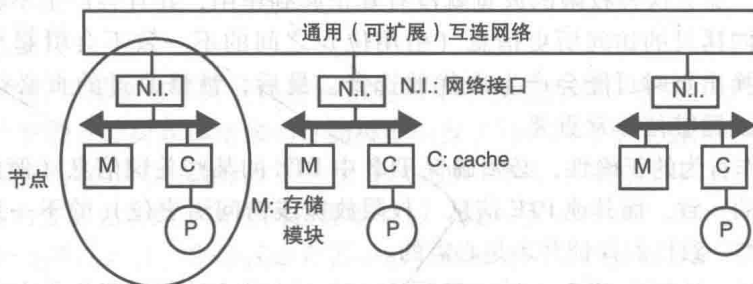


图 5-18 cache 一致性非均匀存储访问架构 (cc-NUMA)

这种模型对片上多处理器也同样适用，在片上处理器中，最高的几级片上存储一般都是处理器的私有 cache，而片上的最后一级 cache（last-level cache），则通常是共享的二级或三级 cache。在这种情况下，从可扩展性的角度来看，可以将共享 cache 分成多个 bank，并将这些共享的 cache bank 分配到各个节点上，从而挖掘存储的局部性。在这种非均匀 cache 访问

(NUMA)结构中,访问本地的共享 cache bank 会比访问远程的 cache bank 要快。

这种将存储模块或共享 cache bank 分配到各处理器节点的内存组织方式,称作 cache 一致性非均匀存储访问架构(cc-NUMA)。在这种结构中,访问处理器本地存储块(或共享 cache bank)和访问远程处理器的存储块(或共享 cache bank)的延迟不同,而且通常访问本地的延迟更短。另外,正如之前所讲的,维护所有节点私有 cache 的一致性是非常有必要的,因此 cc-NUMA 系统也是 cache 一致的。

侦听 cache 协议需要进行改造才能适用于 cc-NUMA 多处理模型。在侦听协议中,在所有事务操作中都需要所有节点的参与,这一点不具有可扩展性。因此在 cc-NUMA 系统中,常常通过目录协议来维护 cache 一致性。

5.5.1 目录协议的基本概念和术语

假设在包含 100 个节点的 cc-NUMA 多处理器系统中,数据块的备份只保存在 2 个节点中。如果采用侦听 cache 协议,那么一个 BusUpgr 请求需要发送给 100 个节点,而实际上相关的节点只有 1 个,因此这会造成带宽的极大浪费。一种更好的解决办法是将一致性事务直接发送给相关的节点,在本例中这种相关节点只有一个。

在 cc-NUMA 多处理器中,数据和代码以页为单位分布在不同的存储模块中。在将虚地址转换为物理地址后,通过物理地址来确定一个读请求真正应该发送到哪个存储模块。假设一级 cache 是物理寻址的,那么这个地址转换过程应发生在访问一级 cache 之前。这个转换过程很简单,如果有 2^n 个存储模块或节点,那么只需要将 n 位——通常是物理地址的高 n 位,作为存放页的存储模块或节点的编号进行索引即可。

cc-NUMA 中常用的一种具有较好可扩展性的协议是目录协议(directory protocol),之所以这样命名,是因为这种协议以内存块为粒度来维护一个目录。每个内存块对应的目录项指向包含该数据块拷贝的所有节点(可能多个),并且记录了数据块在各节点中的状态。

cache 目录协议的设计空间很大,大体上可以分为两类:以内存为中心的和以 cache 为中心的。在以内存为中心的目录协议中,每个存储模块都有一个关联的目录,它记录了本存储模块内每个内存块在节点间的共享信息。另一方面,在以 cache 为中心的目录协议中,通过 cache 项来维护目录信息,cache 项记录了节点间共享块的信息。下面我们首先介绍以内存为中心的目录协议,重点分析实现和性能两方面可能的问题,并探讨其他解决方法。

在以内存为中心的目录协议中,当一个节点发送 BusRd 或 BusUpgr 请求,这个请求会被发送到存储该数据块的存储模块中。在将虚地址转换为物理地址后,通过该块物理地址的某几位来决定选择哪个存储模块。以 BusRd 请求为例,请求块的目录项会指出内存中的备份块是否是最新的,并且可以返回给发送请求的处理器(净 cache 失效),或者是出现脏 cache 失效时,这就需要将请求再前递给某个节点。同样 BusUpgr 或 BusRdX 请求也要先根据物理地址发送到某个存储模块。目录项指出了含有备份块的节点,需要向这些节点发送使之无效的请求。综上所述,一个一致性事务中可能含有 3 种节点:初始化事务的节点,保存目录信息的节点以及含有特定数据块的节点。在本章,我们将初始化请求的节点记为本地(local, L)节点,将保存目录信息的节点记为主(home, H)节点,将其他所有参与事务的节点记为远程(remote, R)节点。注意,主节点可以同本地节点或任何一个远程节点是同一个物理节点,但本地节点和远程节点一般是不同的物理节点。

5.5.2 目录协议实现方法

基准的目录协议一般被称为存在标志向量协议(presence-flag vector protocol)。

基准目录协议所需的硬件结构

存在标志向量记录了包含某个内存块备份的节点集合。图 5-19 给出了结构示意图，并且放大了存储模块目录的具体组织形式。目录中对每个内存块都有一个对应的表项，每个表项包括一个存在标志向量（PFV）以及一个脏位（D）。对于有 n 个节点的系统中，PFV 的长度为 n 位。比如，0 号块的 PFV 是 0110...01，表明第二个、第三个以及最后一个节点有这个块的备份。脏位为 0 表明内存中的块备份是新的。对于 1 号块，只有第二个节点包含它的备份，并且根据脏位为 1 可以知道，cache 中的备份块是系统中唯一更新了的块备份。最后一个块的 PFV 表示所有节点都有这个块的备份，并且由于脏位为 0，这些备份同内存中的备份都是一致的，也就是说，这个内存块是干净的。

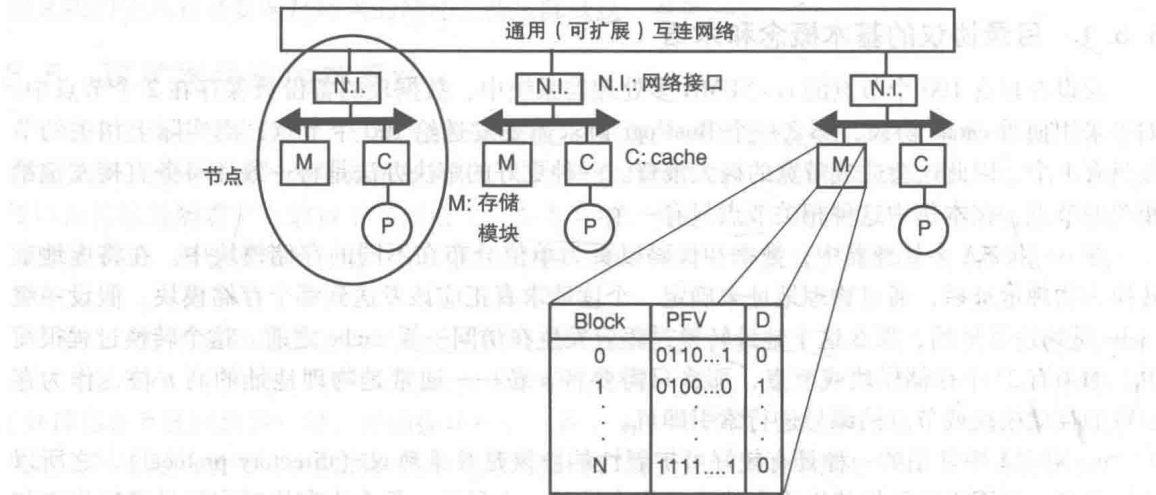


图 5-19 存在标志向量（PFV）目录协议的硬件结构

现在，我们来讨论存在标志向量协议的实现。为了简单，从基于无效策略的 MSI 协议中来探讨实现方法，当然，实际上也可以使用基于更新策略的协议或 MOESI 协议族中的任何一个协议来实现。

在 MSI 协议中，私有 cache 中的每个备份块可能有 3 种状态：被修改过的（M），共享（S）的，无效的（I）。内存中的备份块有两种状态：干净的或脏的。现在我们再回顾一下 BusRd、BusUpgr 和 BusRdX 操作所需的事务，图 5-20 列出了这些事务。

基准目录协议的行为特征

我们从在本地节点（L）的私有 cache 读失效开始，需要考虑两种情况。第一种情况，内存的备份块是干净的。BusRd 请求首先被发送给主节点（H），如图 5-20a（子事务 1）所示。由于该块的目录项中脏位为 0，主节点会以刷新操作进行响应（子事务 2），同时更新目录项，从而反映本地节点有一个块的备份。

如果内存块是脏的，主节点会将该读请求通过一个远程读请求（remote-read request, RemRd）事务前递给远程节点（R），通过目录项可以识别出远程节点（子事务 2），如图 5-20b 所示。当收到读请求后，远程节点会将更新的块返回给主节点（子事务 3）。主节点收到备份块后，将更新内存块以及目录信息，包括清 0 脏位，以反映本地节点现在有一个该块的备份。最后，将备份块发送给本地节点（子事务 4）。

当对处在共享状态的块发送写请求时，可能会出现两种情况：内存中的备份是唯一的备份块；或者多个 cache 中包含此块。第一种情况，如图 5-20c 所示，会向主节点发送 BusUpgr 请求。主节点会更新目录信息，以反映内存中的备份块是脏的。然后内存会向请求节点返回一个

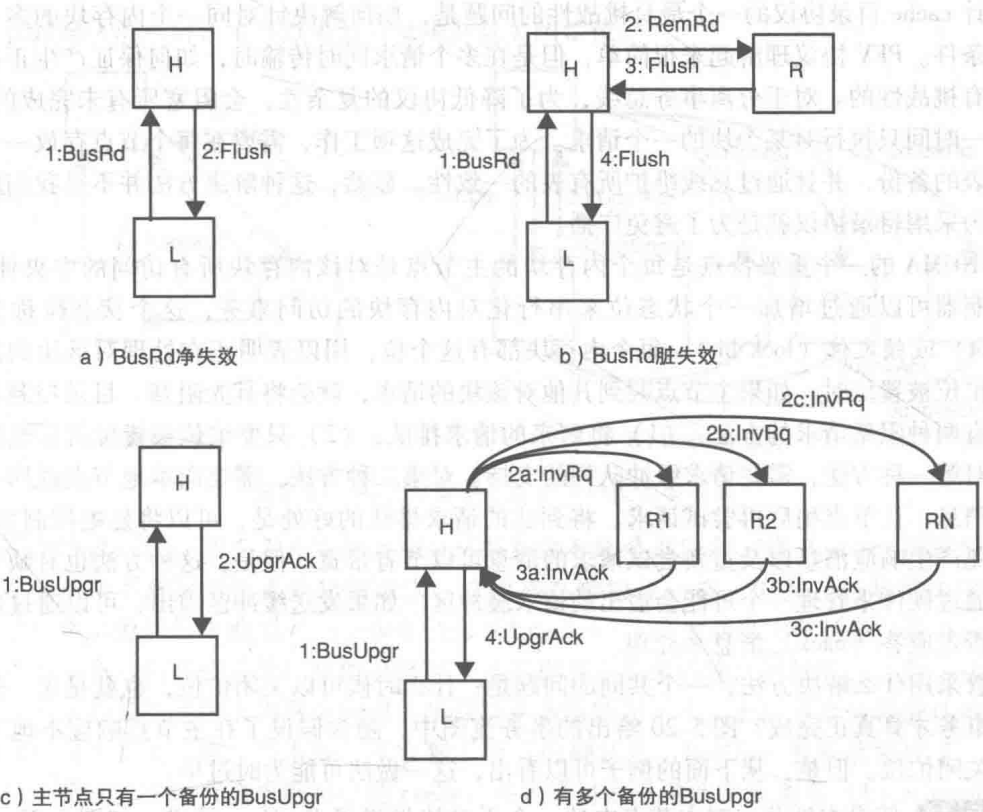


图 5-20 存在标志向量协议的一致性事务

确认，如图 5-20d 中 UpgrAck 所示，这个确认表明事务完成。

如果其他 cache 中也有备份块时，主节点必须对所有包含该块且状态为共享的 cache 发送无效请求，图 5-20c 给出了这一过程。可以通过 PFV 找到包含这些拷贝的节点，假设有 N 个节点都有这个块的备份，则无效化请求 (InvRq) 将发送给 $R1 \sim RN$ ，我们用 2a、2b 和 2c 来表示这些子事务，以表明它们可以并行发出的。当收到无效化请求后， $R1 \sim RN$ 节点向主节点发回确认 (子事务 3a、3b、3c)。当主节点收到所有确认后，将会更新目录信息，表明只有本地节点有唯一的备份，并且内存备份块状态变为脏。最后，主节点通过向本地节点发送 UpgrAck 响应，表明本次更新事务完成。

BusRdX 请求产生的动作同 BusUpgr 请求大致相同。唯一不同的地方在于，在无效化远程节点的备份之前，如果内存中的备份块是干净的，则需要向本地节点发送这个块。如果内存备份是脏的，就需要持有最新备份的远程节点通过主节点将最新的备份块传给本地节点。同 BusRd 请求一样，BusUpgr 和 BusRdX 请求可能会 0 次、2 次或 4 次跨越互连网络。如果本地节点同主节点是一个节点，并且唯一的备份块在内存中，就不需要跨越互连网络。极端情况下，如果本地节点、主节点和远程节点是 3 个不同的节点，就会 4 次跨越互连网络。

为了维护节点间对内存块共享情况的准确信息，当某个块备份从某个节点中被替换出时，也必须通知主节点。如果块是脏的，那么写回的时候自然就通知了主节点。但是，对共享块的替换可能不会通知主节点。这种情况下，存在标志就会不明确，并且存在标志向量指向的节点可能是实际拥有备份块的节点的超集。这种做法不会带来正确性的问题，唯一的问题是可能会由于无用的无效化请求而浪费带宽。归结起来，是更新存在标志，还是发送无用的无效化请求，这需要根据两者的开销和复杂性进行权衡后决定。

设计 cache 目录协议的一个最具挑战性的问题是,如何解决针对同一个内存块的多个请求的竞争条件。PFV 协议理解起来很简单,但是在多个请求同时传输时,如何保证产生正确的行为是很有挑战性的。对于分离事务总线,为了降低协议的复杂性,会阻塞所有未完成的请求,而在同一时间只执行对某个块的一个请求。为了完成这项工作,需要在每个节点存放一个未完成请求表的备份,并且通过总线维护所有表的一致性。显然,这种解决方法并不是我们所希望的,因为采用目录协议就是为了避免广播。

cc-NUMA 的一个重要特点是每个内存块的主节点是对该内存块所有访问的中央仲裁者。目录控制器可以通过增加一个状态位来串行化对内存块的访问事务,这个状态位称为忙位 (busy bit) 或锁定位 (lock bit)。每个内存块都有这个位,用以表明正在处理对该块的某个事务。当忙位被置位时,如果主节点收到其他对该块的请求,就会将其先阻塞。目录控制器对于某个块有两种阻塞请求的方法:(1) 将到来的请求排队;(2) 只要忙位被置位就拒绝到来的请求。对第一种方法,需要请求缓冲队列的支持。对第二种方法,需要向本地节点返回一个否定应答消息,让节点稍后再尝试请求。将到达的请求排队的好处是,可以将延迟控制到最小,因为避免产生响应消息以及重新尝试请求的消息可以节省带宽。但是,这种方法也有缺点,就是需要通过硬件来管理一个可能会溢出的请求缓冲区。如果发送缓冲区溢出,可以通过向请求方发送否定应答 (nack) 消息来处理。

不管采用什么解决方法,一个共同的问题是:什么时候可以关闭忙位,也就是说,什么时候一个事务才算真正完成?图 5-20 给出的事务流图中,隐含假设了在主节点响应本地节点之前可以关闭忙位。但是,从下面的例子可以看出,这一做法可能为时过早。

例 5.11 假设本地节点对主节点中的一个干净块发送了 BusUpgr 请求。如图 5-20c 所示,主节点更新目录,将脏位置位。然后在用 UpgrAck 响应本地节点之前复位忙位。现在假设,在忙位清 0 后,主节点马上就收到了对同一个块的 BusRd 请求。根据图 5-20b 的事务流中向本地节点发送 RemRd 请求,主节点会将新的请求发送给本地节点。如果 RemRd 请求先于 UpgrAck 响应到达本地节点,根据图 5-20c,本地节点将无法处理 RemRd 请求。那么应该如何处理这种情况呢?

问题出在 RemRd 请求先于 UpgrAck 响应达到本地节点。如果两个节点间的子事务可以通过不同的路径传输,就可能出现这种情况。如果两个节点之间只有一条路径,RemRd 就会在本地节点收到 UpgrAck 之后被接收。考虑到无法保证这种顺序性,cache 目录协议必须有能力处理这种竞争条件。一种可能的解决方案是,让本地节点向主节点发送一个否定应答。最终,如图 5-20b,本地节点会收到 UpgrAck 响应,然后处理 RemRd 请求。

降低基准目录协议中的延迟

尽管 cache 目录协议在维护一致性方面,相比侦听 cache 协议可以显著降低带宽,但这是以增加 cache 失效和更新时的延迟为代价的,因为请求都必须先发送给主节点。基准目录协议为了执行一个 BusRd、BusUpgr 或 BusRdX 请求最多会产生 4 次互连网络的跨越。我们现在来讨论一种应用于斯坦福的 DASH 系统的替代方案,DASH 系统是斯坦福大学在 20 世纪 90 年代初设计的一个非常有影响力的研究原型机,它可以降低事务的延迟。

图 5-21 给出了这个替代方案中 BusRd 和 BusUpgr 的事务流,将 4 跳事务变为了 3 跳事务。BusRd 事务给出了当内存块备份是脏时的事务流。当收到本地节点的 BusRd 请求后,主节点开启忙位,并通过一个 RemRd 请求将这个请求发送给远程节点,这几步同图 5-20 中基准协议是一样的。当收到 RemRd 请求后,远程节点会做两件事:将块传送给本地节点(子事务 3a),同时并行地向主节点返回一个确认 (RemAck) 消息(子事务 3b),以通知主节点事务已经成功

完成。收到确认消息后，主节点更新目录项并且关闭忙位。

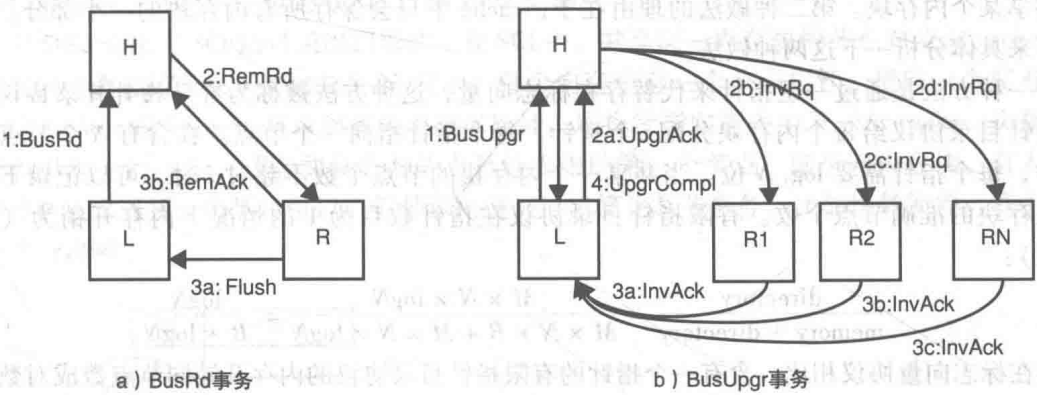


图 5-21 替代方案中三跳目录协议的事务流

同基准协议一样，图 5-21b 中的 BusUpgr 事务由本地节点发起，将 BusUpgr 请求发送给主节点。主节点接下来发出使无效化的请求，这也和基准协议一样。但是，与基准协议不同的是，主节点也会返回给本地节点一个响应消息 (UpgrAck)，这个消息中包括备份块节点数目的信息。本地节点可以利用这个计数值对接下来从远程节点传来的确认消息进行计数。这个消息很重要，为了在最坏情况下将网络跨越的步数从 4 步降低到 3 步，远程节点会将确认消息发送给本地节点而不是主节点。当本地节点收到了所有的确认后，BusUpgr 事务就完成了。整个事务的最后一步是，本地节点更新自己的 cache，并且通过发送更新完成消息 (UpgrCompl) 通知主节点事务已经完成。当收到这个消息后，主节点会更新该块的目录项并且关闭忙位。

5.5.3 目录协议的扩展性

基准目录协议的存储需求

基于存在标志向量的 cache 目录协议，有一个缺点——它需要大量存储来保存目录项。假设每个节点的内存含有 M 个块，每个块的大小是 B ，多处理器含有 N 个节点，存在标志向量需要的存储大小（除去脏位）为：

$$\text{directory} = M \times N^2$$

因此，目录的大小与节点数呈平方成正比。

例 5.12 一个基于存在标志向量的 cache 目录协议需要维护一个包含 128 个节点的多处理器一致性。块大小为 16 字节。那么，为了实现目录需要占用的存储比例为多少？忽略脏位。

实现目录需要占用的存储比例是每个节点的块数 (M)、块大小 (B) 和节点 (N) 的函数，下面给出它们的关系：

$$\frac{\text{directory}}{\text{memory} + \text{directory}} = \frac{M \times N^2}{M \times N \times B + M \times N^2} = \frac{N}{B + N}$$

在本例中，实现目录所占的存储比例为 $128 / (128 + 128) = 0.5$ 。表明 50% 的内存要用来存储目录项。也就是说，保存目录所用的内存空间同保存数据所用的空间一样大！

很明显，基准目录协议并不具有良好的扩展性。即使对那些节点数适中的系统也是如此，这种做法的内存开销太大了。当然，我们可以通过增加块大小来降低内存开销，当块的大小为 64 字节时（这是比较合理的大小），开销会降低到 12.5%。但是为了获得可接受的更低开销，有必要去开发一些其他的目录实现方法。

其他具有更低内存开销的目录实现方法

可以通过两种方法来降低存储目录的内存空间：每个节点的状态位小于 1 位，或者每个节

点内的目录项个数小于数据块数。第一种做法的理由在于,在所有节点中,只有一小部分节点会共享某个内存块。第二种做法的理由在于,cache 中只会保存所有内存块的一小部分。我们接下来具体分析一下这两种做法。

一种方法是通过一组指针来代替存在标志向量,这种方法被称为有限指针目录协议。有限指针目录协议给每个内存块分配 i 个指针,每个指针指向一个节点。在含有 N 个节点的系统中,每个指针需要 $\log_2 N$ 位。当共享一个内存块的节点个数不超过 i 时,可以记录下共享该内存块的准确节点个数。有限指针目录协议在指针数目为 1 的情况下内存开销为(忽略脏位):

$$\frac{\text{directory}}{\text{memory} + \text{directory}} = \frac{M \times N \times \log N}{M \times N \times B + M \times N \times \log N} = \frac{\log N}{B + \log N}$$

同存在标志向量协议相比,含有一个指针的有限指针目录协议的内存开销同节点数成对数关系增长,而不再是线性增长。然而,有限指针目录协议的一个重要问题是,如何处理共享节点数超过有限的指针数目,换句话说,如何处理指针溢出。

对于指针溢出的一种解决办法是,当目录指针用尽时,采取广播的方法。这种做法被记为 $\text{Dir}_i B$, B 表示广播, i 是指针的个数。例如,对于 $\text{Dir}_4 B$ 协议,当第 5 个节点发生 cache 失效时,该项的广播位被置位,表示下一次发生对该块的写时,会向所有节点发送使无效化请求。另一种解决办法是,在大规模系统中通过指针替换来避免广播。当目录项有 4 个指针可用时,如果有第 5 个节点访问该块时,选择当前指针指向的某个节点作为替换物。这个替换节点的备份会被无效化,这样就可以为新的节点分出一个指针。

MIT 的 Alewife 多处理器系统采用了一种与上面方法截然不同的指针溢出处理方法,硬件提供了有限个指针,而指针溢出的问题则交给软件处理。这种方法称为 $\text{Dir}_i \text{SW}$ 。当“硬件”指针被用完时,会触发一个软件处理程序的陷阱,然后处理程序在常规(非目录)内存中给新指针分配位置。这种模式不仅在硬件指针不够用而有新的节点需要拷贝块时依赖于软件程序,并且当某个块被修改后,软件已分配过一些指针,都需要依靠软件来处理。通常情况下,如果共享块的节点数少于硬件指针数时,有限指针模式是非常高效的。

存在标志向量和有限指针两种模式都是为了维护节点中块拷贝的准确信息。另一种降低内存请求的替代方法是将节点分组为集群或区域。例如,一个 128 个节点的多处理器系统按照每个集群含有 4 个节点分组,则存在标志向量需要记录 32 个集群的存在信息,而不再是 128 个节点的信息,这就使得目录对内存的需求减少到 1/4,这种模式也被称为粗向量目录协议。

我们也可以设计一种混合模式,初始时像有限指针模式一样,当硬件指针消耗完后,实现指针的那些位的语义将改变为对粗向量目录的编码。例如,在一个拥有 128 个节点的多处理器系统中有 4 个硬件指针,每个指针需要 $\log_2 128 = 7$ 位,4 个指针就需要 28 位。在 32 位的目录项中,每个目录项可以编码 32 位的存在标志,每一位对应一个包含 4 个节点的集群。

到目前为止,减少目录大小的方法还仅限于减少每个目录项的位数。但是,目录项的个数其实也是可以降低的。这类降低目录对内存需求的方法基于以下观察结果:在任何时刻,cache 中保存的块的个数远小于内存中的所有块数,这也就是说,大多数目录项是没有被用到的。这也是目录 cache 技术的理论依据。在目录 cache 中,当某个新节点需要某个块时,才会分配目录项。当目录 cache 中某项被替换时,整个系统中的这个换出块都会被无效。如果内存中的块数是 cache 行数的 1000 倍,相比于存在基准标志向量协议而言,所需要的内存就会降低 3 个数量级!这样做的缺点就是,有时会发生目录 cache 的失效。

以 cache 为中心的目录协议

将目录分布在 cache 而不是内存中,这样目录项的数目自然就是 cache 行数而不是整个内

存中的块数。以 cache 为中心的目录协议的一个典型例子就是可扩展一致性接口 (Scalable Coherent Interface, SCI)，这也是 IEEE 1596-1992 标准。

图 5-22 给出了 SCI 的目录组织形式。在 SCI 中，共享同一内存块的节点对应的 cache 目录通过双向链表连接起来。正如图 5-22 所示，每个内存块有一个节点指针，指向一个拥有该块备份的 cache。每个 cache 目录项有两个节点指针，尾指针指向链表中下一个 cache，头指针指向链表中前一个 cache，或者如果当前节点是链表中的第一个节点，则指向内存块。有人提出使用单链表来连接，虽然也可以，但是双向链表可以简化某些事务，比如块替换等，我们接下来会进行讲解。

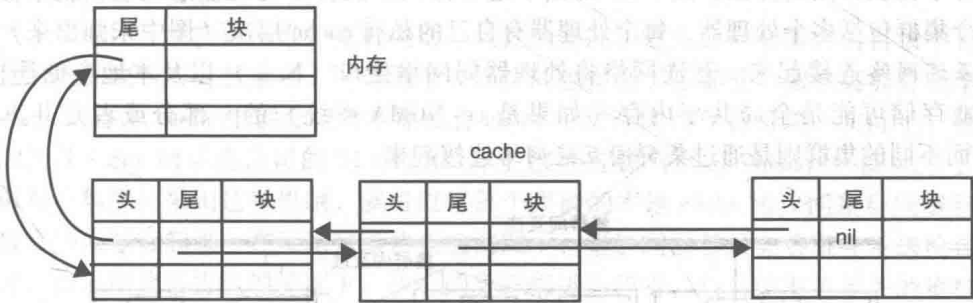


图 5-22 可扩展一致性接口 (SCI) 目录组织

我们依次来回顾一下协议事务中 BusRd、BusUpgr 和替换请求这几个事务。BusRd 事务是从一个节点向主节点发送 BusRd 请求开始的，会出现两种情况。如果没有 cache 有该块的备份，主节点就会通过两跳事务将块传给本地节点，并且通过将尾指针指向本地节点来将本地节点加入链中。如果其他 cache 有备份块，就会将新共享节点插入主节点和旧的头节点之间。主节点首先会向旧的头节点发送请求，使其头指针指向链中的新节点——新的头节点。它同时将头节点的标识传给新的头节点。然后新的头节点将它的尾指针指向旧的头节点。

BusRdX 或者 BusUpgr 请求的动作是以将请求节点插入链表头部开始的。如果请求节点没有块备份 (BusRdX 的情况)，BusRd 事务首先会将请求节点插到链表头部。接下来，请求节点会向它的后继节点发送使无效请求，后继节点又会将此请求向后传。这个动作会一直递归地执行，直到链中其他节点的相应块都被无效了。当无效化请求传播到链表中的最后一个节点时，最后一个节点通过内存向头节点发送一个确认消息，以表明写操作已经完成。另一方面，如果请求节点有块的备份 (BusUpgr 的情况) 并且节点处在链表中间，那么它必须先移动到链表头部。为了完成移动，要将节点从链表中摘除，然后再插入链表头部。将节点从中间移到头部需要采取同移出块一样的步骤。

当 cache X 中的某个块备份在链表中间，并且这个块要被移出时，双向链表按照如下方法更新：cache X 发送一个包含被移出块尾指针的消息给它的头指针指向节点，这样这个节点就可以直接连到 X 尾指针指向的节点。同样 cache X 向它的尾指针指向节点发送一个它的头指针，这样尾指针指向的节点就可以指向 X 的头指针指向节点了。

现在我们比较一下 SCI 和 PFV 协议在延迟、带宽和内存占用方面的表现。对于 BusRd 事务，SCI 和存在标志向量协议的延迟不分上下，因为，在净失效的情况下两种情况都需要一个两跳事务来提供数据。对于总线更新事务，SCI 的延迟与共享块的节点个数成正比关系，因为使无效化请求串行地遍历共享链表。这与存在标志向量协议不同，在 PFV 中，使无效请求是并行发出的。由于含有备份块的节点数量一样，SCI 和存在标志向量协议在维护一致性时

消耗的带宽是相同的。因此，在两种协议中，BusUpgr 事务消耗的带宽都是与含有备份块的节点的个数成正比的。SCI 和 PFV 在内存占用方面不同，在 SCI 中，每个内存块有两个指针，每个 cache 块有两个节点指针，而存在标志向量中每个内存块需要 N 位来保存向量。这表明，SCI 的内存开销是同 cache 行数成正比的，而不是同内存块数成正比，这样就可以节省很多空间。

5.5.4 层次化系统

从单处理器节点的平坦组织结构出发，设计一个包括多个处理器的节点的可扩展系统，是现在设计中比较常用的方法，图 5-23 给出了这种方式。这种多处理器系统由一组集群节点构成，每个集群包括多个处理器，每个处理器有自己的私有 cache 层次（图中未画出来）并通过集群内互连网络连接起来。互连网络将处理器同网络接口（N.I.）以及本地存储连接起来，这些本地存储可能是全局共享内存（如果是 cc-NUMA 系统）的一部分或者是共享的集群 cache。而不同的集群则是通过集群间互连网络连接起来。

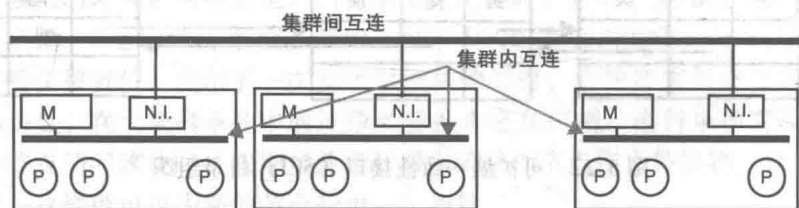


图 5-23 两级多处理器系统组织方式

很多商用机都采用了类似的层次化存储组织形式。20 世纪 90 年代非常成功的大型系统 SGI Origin 2000 就是由多节点构成的，每个节点包括两个处理器。Sun 微系统公司的 WideFire 原型机由 4 个 E6000 节点（每个含有 30 个处理器）连接成为一个具有 100 多个处理器的系统。目前片上多处理器的集成密度已经非常高，这也使得将来可能会进一步提高大家对层次化系统的研究热情。随着包含数十个处理器的片上多处理器成为现实，只用一小部分这些芯片就可以组成包含几百个处理器的低开销多处理器系统。

根据集群的数量以及集群间的互连网络类型，我们可以选择通过侦听 cache 协议或者 cache 目录协议来维护集群间的 cache 一致性。在节点内部也是类似，我们可以根据节点内互连网络的类型来选择侦听协议或者目录协议。在介绍完这些协议的具体细节之后，现在需要考虑的一个现实问题是，如何将这两种协议（一种用来维护集群间的一致性，一种用来维护集群内部的一致性）结合起来维护整个系统所有处理器的 cache 一致性。我们给出 3 种可选的方案：集群间和集群内部都采用侦听协议；集群内采用侦听协议，集群间采用目录协议；集群内采用目录协议。

集群内部和集群间都采用侦听协议

首先来考虑集群内部和集群间都通过侦听来维护一致性的情况。BusRd 或者 BusRdX 请求都必须传送给系统中的所有处理器。首先，请求被发送到本地集群的内部互连网络上。如果本地集群可以满足这个请求，那么事务就可以完成；如果本地集群无法满足该请求，事务就会被发送到集群间互连网络，传送到其他所有集群。接下来，整个系统的所有私有 cache 都需要侦听这个请求。但是，为了节省集群内部宝贵的带宽，可能会增加一个很大的集群 cache 并且集成到网络接口上。对于片上多处理器的集群 cache 的选择，经常会选择二级或者三级 cache。

如果保证了集群 cache 和集群内部私有 cache 之间的包含关系,那么集群 cache 会过滤掉大部分对本集群的请求。

集群内采用侦听协议,集群间采用目录协议

在这种情况下,集群间的目录协议会将每个集群看作一个单独的节点。集群间的目录协议会保存集群对于每个独立的内存块的共享情况。正如之前讲解的那样, BusRd 以及 BusRd 请求会被发送到主节点的存储模块,而主节点的存储模块保存了涉及该事务的集群。当一个集群收到了 BusRd 或者使无效化请求,它会将这个请求发送到集群内部的互连网络,根据节点采取的侦听 cache 协议,每个私有 cache 都会侦听本地互连网络。

集群内采用目录协议

最后,考虑用目录协议来维护集群内部一致性的情况。首先,一个集群的目录中只包含集群中缓存的内存块对应的目录项,这样既节省了空间也保证了正确性。如果集群有共享的最后一级 cache,则这个 cache 中就会记录本地私有 cache 中缓存的内存备份块。实际上,这是一个很常用的选择。Sun 微系统公司的 T1 SPARC 片上多处理器(代号 Niagara)就采用了这种方法。如果每个集群都采用这种机制,就可以将每个集群的本地 cache 协议同集群间级别的侦听协议或目录协议结合起来。在采用目录协议的情况下,集群间的目录会将事务发送给包含请求块的集群。在采用侦听协议的情况下,所有的集群都进行侦听,与当前事务无关的集群就会通过集群 cache 来过滤掉这个事务。

层次化系统可以减少目录协议的存储开销,我们通过下面的例子进行说明。

例 5.13 假设某个系统包含 128 个节点,内存块大小为 16 字节,采用基于目录的协议来维护私有 cache 一致性。如例 5.12 所示,对于存在标志向量协议的内存开销是 50%。如果采用集群的结构,每个集群包括 8 个处理器,集群内通过侦听协议维护一致性,集群间通过存在标志向量协议维护一致性,那么需要的内存是多少?

如果每个集群有 8 个节点,那么就会有 16 个集群,每个目录项需要 16 位。因此内存开销是 $16/(128 + 16) = 11\%$,这个内存开销比之前的单级组织方式看起来要合理多了。

5.5.5 页面迁移和复制

对可扩展性的需求促使我们在 cc-NUMA 多处理器系统中将内存分布到各个节点。在有私有 cache 的情况下,页替换策略就显得不那么重要了。但是私有 cache 容量有限,对于那些工作集比私有 cache 还大的应用程序来说,容量失效就会很高。为了降低 cache 失效的开销,一个节点经常访问的数据和指令在理想情况下应该放在本地内存或本地共享 cache 中。由于数据结构在内存中是以页为单位分配的,为了进一步开发局部性,虚存管理系统就显得格外重要了。

下面给出了几种可供选择的方法来将以页为单位的数据块存放到不同存储模块中。页可以按照静态方式进行映射和存放(即静态页面布局),或者也可以根据应用程序的访问模式在程序运行过程中动态地改变页面的存放(即动态页面布局)。当任务在各节点的分布在运行时改变,或者我们运行程序之前无法预测它被分布到哪些处理器上时,采用动态页面布局就十分合理。通过下面矩阵乘法的例子,我们可以看出各种页面布局方法出现的动机。

例 5.14 回顾图 5-2 给出的矩阵乘法的并行实现。图 5-24a 从图 5-1 借鉴了矩阵乘法的串行实现。结果矩阵的每个元素 $C[i, j]$ 是通过矩阵 A 的第 i 行和矩阵 B 的第 j 列的点积操作得到的。这步是通过索引为 k 的内层循环完成的。

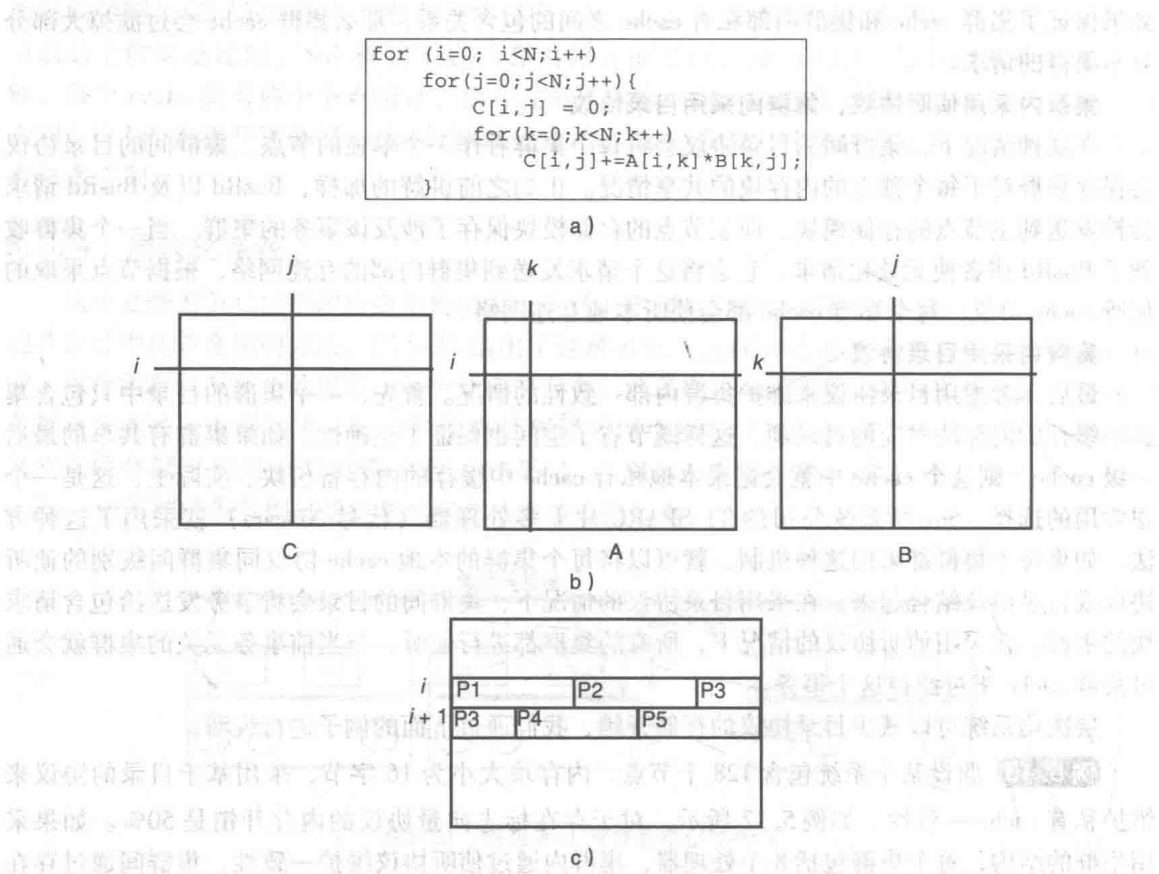


图 5-24 矩阵乘法的并行实现 (a) 索引模式 (b) 和页面布局 (c)

并行实现是将算法最外层的 for 循环进行并行。假定矩阵在内存中按行优先存储，所有矩阵的行在内存中都是一行接一行存储的。图 5-24c 第 i 行占据了两页 (P1 和 P2)。第 3 个页 (P3) 是两个连续行 i 和 $i+1$ 共享的。任务大致被静态分到 P 个线程，矩阵 C 的前 N/P 行是由第一个线程计算的，接下来的 N/P 行由第二个线程计算，依此类推。矩阵 A 和 C 的前 N/P 行只被第一个线程访问，接下来的 N/P 行只被第二个线程访问，依此类推。在这个例子中，采用静态页面布局的方法会给并行矩阵乘法带来很大的好处。比如，将所有包含矩阵 A 和 C 的前 N/P 行的页面分配到第一个线程的本地内存中。通过这种静态的划分可以提高局部性。但是，静态任务分配并不总是好的，甚至有时候是不可行的。以矩阵乘法为例，出于工作负载均衡性的考虑，可能希望将矩阵的行动态地分给各线程。这样，显然静态页面布局就不如动态布局有效了。此外，如果允许进程从一个处理器迁移到另一个处理器，页面也应该被允许迁移。本节讲述的一类重要的动态页面布局机制称为页面迁移机制 (page-migration scheme)。

回到矩阵乘法的并行实现中，我们发现无论怎样将矩阵 A 和 C 的行分配到各线程，所有的线程都需要访问矩阵 B 的所有元素，也就表明无论采取静态或动态的页面映射，都只会对其中的一个线程有利。因此需要做的是将包含矩阵 B 的元素的页面在所有线程中存储模块都复制一遍。页面复制主要受到两个因素的制约：第一，物理页面的复制会消耗宝贵的物理内存空间，因此需要在挖掘局部性和降低缺页率之间进行权衡；第二，如果在内存这一级没有维护一致性，那么只能复制只读的页面。

那些很少被虚存管理程序修改的页面也可以进行页面复制。起初,一个页面可能是只读并且写保护的,当这个页面被修改时,内存控制单元通过写访问保护异常处理程序检查到这一修改操作,并且在处理程序中采取恰当的动作,这些动作包括使其他节点中对该页的 TLB 项无效,并且使该页对应的 cache 项无效,这同 5.4.7 节中讲的一样。在稍后的某个时间,可以通过类似的机制将访问权限转回为只读的。这种允许一个虚页有多个拷贝的物理页布局机制也被称为页面复制机制 (page-replication schemes)。

静态页面布局机制

在不知道线程对页面的访问模式的情况下,一个合理的策略就是将这些页面平均分配到各节点。一种方法叫作轮询 (round-robin) 页布局。在这种方式下,假设有 N 个节点,则页面 K 被映射到节点 $L = K \bmod N$ 上。有些操作系统,比如 SGI 的 IRIX,允许在代码中嵌入制导命令来选择页面布局方式。它们也支持利用代码中的指令将某个页固定到某个节点上,我们可以利用这个特点,在矩阵乘法例子中将矩阵 A 和 C 的页面固定住。

另一种静态页面布局策略是首次访问 (first-touch),即页面被分配到第一次访问它的处理器的内存中。根据直觉,第一个访问页面的线程可能是唯一访问该页的线程,或者至少是访问最频繁的线程。对于上面提到的矩阵乘法的例子,首次访问布局会根据任务分配情况进行页面分配,这是有利的。但是,总的来说,首次访问页布局也有缺陷。比如,很多并行程序在进入并行阶段之前会有初始化的阶段,在初始化阶段,主线程会为并行阶段进行数据准备,这就可能从磁盘中读取原始数据。例如,在矩阵乘法的算法中,主线程可能会先读取矩阵 A 和 B 的内容。如果没有特殊的处理,首次访问策略可能会将主线程访问的所有数据都放到主线程所在的处理器节点上。在这种情况下,在进入并行阶段前采用首次访问进行页面分配是不会有好的效果的。为了解决这个问题,可以在代码中插入指令来保证只有进入并行阶段后才能采用首次访问进行分配。

指导页面分配策略的一个常见问题是页面本身的粒度大小。比如,矩阵的一行大小很可能不是页面大小的整数倍。这样,一个页面可能被多个线程访问。在矩阵乘法的例子中,如图 5-24c 所示,第 i 行和第 $i+1$ 行共享的页面 $P3$ 就是这种情况。

页面迁移机制

假设我们提前知道了每一个线程的访问模式,在这种情况下,就很有可能对每个页决定最优的静态分配:只需要对每个页面访问的线程进行计算,将这个页面分配到访问次数最高的节点上。页面也可能出现迁移,从而增加对每一页的本地访问。如果知道页面迁移的访问开销,就可能针对是否迁移以及什么时候进行动态迁移得到最优决定策略。

例 5.15 假设对一个页远程访问的开销是 C ,迁移一个页的开销是 M 。为了简便,假设本地内存访问的开销是 0。现在考虑这种情况,一个分配到 $N1$ 节点的页被另一个节点 $N2$ 上的一个线程访问了 K 次。那么这个页要被 $N2$ 访问多少次才能抵消掉迁移到 $N2$ 的开销?

如果没有迁移,访问开销是 $K \times C$ 。如果进行页迁移,开销的下界是 M 。那么,当 $K > M/C$ 时就可以抵消掉页迁移的开销。

然而实际情况是,在大多数情况下,我们是无法提前知道尚未发生的访问序列的。因此,在线(或动态)页面迁移机制必须根据之前的访问序列来预测未来的访问序列。一种方法是记录远程访问的次数,当次数超过某个预先设定好的阈值时,就进行页面迁移。在例 5.15 中,当 K 超过 M/C 时,就可以进行页面迁移了,这就将总开销限制到了 $2M$ 。为了实现这种机制,在有 N 个节点的系统中,每个页面需要 N 个计数器。在程序运行前,每个页面的计数器都置为 0。当远程访问一个页面时,与远程节点对应的计数器就会加一,其他某个任选的计数器被

减一。当计数值超过 $2M/C$ ——迁移开销的 2 倍（以远程访问开销为单位）时，就将这个页迁移到远程节点。

采用这种方法，访问的开销不会超过预先知道未来访问序列的最优方法的 3 倍。举个例子，如果节点 N1 的一个页被节点 N2 远程访问了 $2M/C$ 次，然后被迁移到 N2，然后又被节点 N1 远程访问了 $2M/C$ 次，再次被迁移到节点 N1，最优迁移算法的开销是 $2M$ ，而在线迁移算法的访问开销是 $6M$ 。对于这种最坏情况下的访问，开销是最优页面分配的 3 倍。

页面迁移开销 M 包括两方面：一个开销是重新映射页面并且改变页表项，同 5.4.7 节中讨论的一样，这会引起 TLB 无效操作；另一个开销是物理地将页面的内容从一个节点的内存移动到另一个节点的内存。

页面复制机制

如图 5-24 所示的矩阵乘法的并行实现中，所有线程都要访问矩阵 B 的所有元素。此时页面迁移机制只会使一个节点收益，因此也就没什么效果，我们需要将所有的页面都复制给每个节点。

一种简单的页面复制机制实现是，对每个页帧提供一组计数器。与页面迁移类似：如果复制开销是 R ，页面在被远程节点访问的次数超过 R/C 次后就进行复制。如果远程节点在复制之后不再进行页面访问了，那么同不复制页面相比，额外的开销就是 R 。

5.6 全 cache 共享内存系统

在可扩展的 cc-NUMA 系统中，更多的是应用基于目录的 cache 协议，因为这种协议只需保持涉及某个一致性事务的节点间的一致性，而不像侦听协议那样需要涉及所有的节点。在 cc-NUMA 设计时另一个很重要的问题是通过静态还是动态页面布局来优化系统，使其在页面级可以尽可能地利用局部性。但是，页面迁移或者复制机制只能以页面为单位来挖掘局部性。只有知道哪个节点对整个页面的数据访问次数最多，静态的或可迁移的页面布局策略才会有用。对页面的复制要求页面经常被访问但很少被修改，因为维护页面级的一致性需要虚存管理系统的参与，这个开销是非常大的。

这些缺点告诉我们，通过硬件协议，以一个更加恰当的单位，比如 cache 块，来支持存储级的迁移和复制机制应该是个不错的想法。本节，我们介绍全 cache 存储体系结构（Cache-Only Memory Architecture, COMA）的概念，COMA 将运行内存块在私有 cache 和本地内存之间迁移和复制。

5.6.1 基本概念、硬件结构和协议

利用对虚存管理系统透明的硬件协议，COMA 系统支持内存块在 NUMA 组织结构中的存储体之间进行迁移和复制。概念上，是将 NUMA 机器的每个节点都转化为一个类似于 cache 的结构，然后利用与维护私有 cache 间一致性类似的机制来维护这些节点间的内存块一致性。

我们先通过基于总线的结构来认识一下 COMA 所需的硬件结构，以及这种共享内存系统所需关注的一些新问题。

COMA 架构的硬件结构和设计问题

为了介绍 COMA 架构，我们先来考虑如何在一个简单系统中支持这种架构，这个简单系统由一些节点组成，而每个节点有一个处理器以及附属的私有 cache 和本地内存。节点间通过基于广播的互连网络（如总线）连接，图 5-25 给出了这种架构。

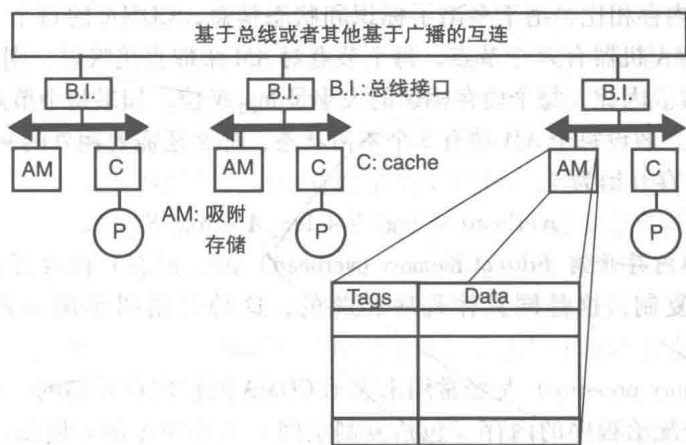


图 5-25 基于总线的全 cache 内存架构 (COMA)

这个多处理器的内存组织方式与图 5-18 所示的 cc-NUMA 内存组织方式不同，这里的每个存储模块被转化为一种类似 cache 的结构，称为吸附存储 (Attraction Memory, AM)。图中给出了放大的 AM 的视图，像 cache 行一样，每个内存块都有一个标识 (tag) 和一些状态信息。利用这些额外的信息，可以将一个块迁移到任何 AM 中，也允许在多个 AM 中有备份。COMA 中的所有 AM 都包含了整个应用程序的足迹，并且有额外的空间来复制一些节点间的内存块。

在 COMA 中每个内存块可以被放在任何节点的内存中，而在 cc-NUMA 中，内存块会固定放在一个有物理地址指向的主位置，这就使得以内存块为单位迁移和复制内存成为可能。虽然便于理解，但将内存视为 cache 会带来一些新的设计问题。

在 cc-NUMA 的私有 cache 中，当一个内存请求失效时，通过物理地址可以找到存储内存块的存储模块，即该块的主节点。但是，在 COMA 中，一个内存块可以存在任何 AM 中。因此，确定块在内存中的位置是很重要的。对块定位 (block localization) 机制的需求是 cc-NUMA 和 COMA 机器的第一个不同。

由于在 COMA 中，本地内存被转化为一个大的 cache 节点 (AM)，那么，在本地 AM 中就可能找到所需的内存块。因此，块定位过程首先通过标识 + 状态检查来遍历本地 AM。如果在本地 AM 中没有找到需要的块，则它一定在某个远程节点中。通过类似于 MOESI 的内存级一致性协议，可以定位该块。一旦定位完成后，本地 AM 中会存入一个备份，这一操作可能会触发 AM 的替换操作。在传统的内存组织的 cache 中，替换一个块很简单，如果没有修改过就可以直接被丢弃；如果修改过，就将它写回到固定的主节点的内存中。但是，在 COMA 中没有这个固定的主位置。如果要换出的块是整个存储系统中唯一的备份块，即使没有被修改过也不能丢弃它。因此，内存级的协议必须能够为这个块找一个新的主位置。块重定位 (block relocation) 是 COMA 机器设计的另一个需要处理的问题。

AM 块替换的频率取决于 AM 的大小以及它们之间的关联性。在传统存储系统中，如果一个程序占用的地址空间与它的整个足迹 (程序执行期间访问的所有内存地址) 的大小一样，则所有的内存块都会有一个固定的主位置。但是，在 COMA 中并不是这样的。正如 cc-NUMA 的页面复制，COMA 中内存块级别的复制需要消耗比应用程序足迹更多的内存空间。因此，程序在 COMA 中会占用比在 cc-NUMA 中更多的内存，内存开销 (memory overhead) 是 COMA 中需要考虑的一个重要问题。AM 中的冲突失效也会触发块重定位。正因如此，还需要慎重考虑 AM 间的关联性。更高的关联性会降低块重定位次数，但也会导致更长的 AM 访问时间。在 COMA 中，块重定位次数、内存开销和 AM 访问时间是设计时需要权衡的重要方面。

同 cc-NUMA 的内存相比，由于多出了标识和状态信息，COMA 的每个 AM 都会增加内存开销。假设一个 COMA 机器有 N 个节点，每个节点对 AM 采取直接映射，则一个内存块可以存放在 N 个不同的地方。因此，每个内存标识的大小是 $\log_2 N$ 位。如果每个节点中的相联度是 A ，标识会增加 $\log_2 A$ 位。假设每个 AM 项有 S 个不同状态，那么还需要额外的 $\log_2 S$ 个状态位。最后，每个 AM 块的内存开销为

$$\text{overhead} = \log_2 N + \log_2 A + \log_2 S$$

除去这个被称为直接内存开销 (direct memory overhead) 的 (静态) 内存开销之外，内存开销的另一个来源是块复制，这是同具体程序相关的，这种开销叫作间接内存开销 (indirect memory overhead)。

内存压力 (memory pressure) 是经常用来表示 COMA 间接内存开销的一个度量，它的大小是由程序足迹除以分配给程序的内存 (包括复制空间) 大小决定的。例如，内存压力为 75% 表明有 25% 的空间可以用来复制。当内存压力增加至接近 100% 时，内存块的替换会增加，性能会下降到发生虚存系统中常见的颠簸 (thrashing) 状态。

例 5.16 假设一个 COMA 机器有 128 个四路组相联的 AM，支持内存级的一致性协议，每个 64 字节的块有 8 个状态，计算总的内存开销 (直接的和间接的)。假定 COMA 的内存压力为 90%。

每个内存块的直接内存开销为 $\log_2 128 + \log_2 4 + \log_2 8 = 12$ 位。内存块大小为 $64 \times 8 = 512$ 位。内存压力为 90% 表示复制需要额外 10% 的空间。因此，为了复制，每个 64 字节的块需要增加 6 字节 = 48 位。最后，这个 COMA 系统的每个块需要 $12 + 48$ 位开销，内存开销的比例为 $60 / (512 + 60) = 0.1$ 。从而，COMA 系统中 10% 的内存空间用来管理 AM 和复制。我们在本例中没有计算目录开销，因为绝大多数可扩展系统都有这个开销，并不只是 COMA 有的。

5.6.2 平坦 COMA

COMA 架构的多处理器系统可能是基于总线的或是分层的，也可能在分布式共享内存多处理器系统 (NUMA) 中通过适当的结构修改来引入 COMA 的概念。图 5-26 给出了平坦 COMA 所需的硬件结构及其组织方式。“平坦” (flat) 指的是存储系统并没有采用层次化结构。

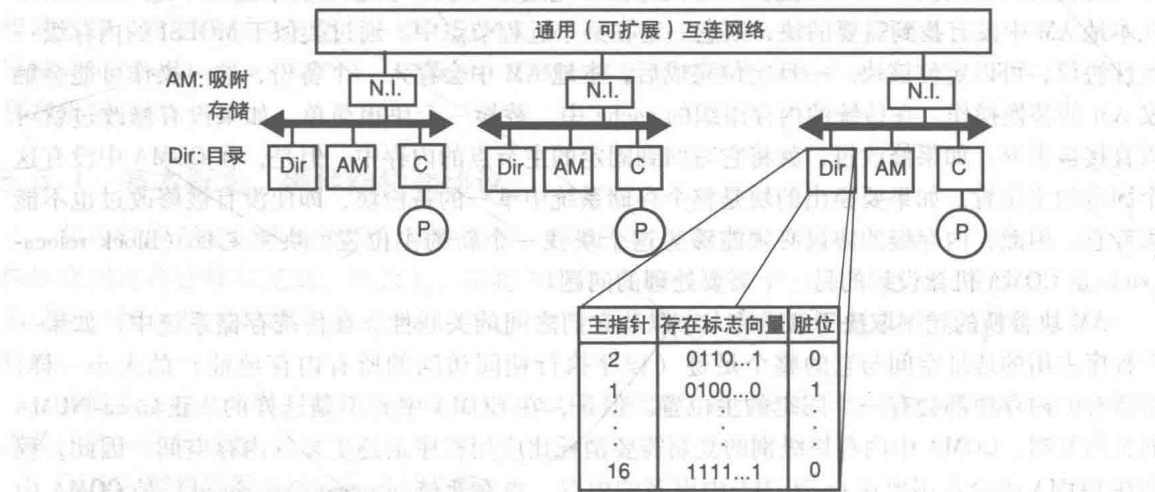


图 5-26 平坦 COMA 的硬件结构和组织方式

同 cc-NUMA 架构类似，平坦 COMA 有多个节点，每个节点有一个或多个处理器以及私有

cache、一部分内存和一个目录结构。与 cc-NUMA 不一样的地方在于,对应每个节点上的共享内存是被组织成吸附存储的结构。我们之前讲过,在 cc-NUMA 中,本地目录中记录了本地节点中的块在其他节点中的备份情况。而在 COMA 中,内存块并没有一个固定存储的主节点,因此需要采取一些不同的方法。

在平坦 COMA 中,内存块对应的目录项有固定的主位置,这同 cc-NUMA 一样,可以通过物理地址定位到主节点。但是,在 COMA 中,主节点的目录只记录含有备份块的远程吸附存储,而不像 cc-NUMA 那样记录的是备份块,而目录项与本地 AM 中的项并不直接相关。图 5-26 中给出了目录结构的详细信息,这里简要介绍一下相关的协议,该协议很大程度上受到了 MOSI 协议的启发, MOSI 协议是 MOESI 协议的子集,它去掉了独占状态,内存块的状态有无效 (I)、共享 (S)、拥有 (O) 和修改 (M) 这 4 种形式。

主节点中的每个目录项有 3 个域,从左到右第一个为主指针 (master pointer),指向状态为 O 或 M 的数据块拥有节点。下一个域是存在标志向量 (PFV),指向 AM 中含有该块的节点。最后一个域是脏位 (D),表明在只有一个备份块的情况下,该块的状态是 O ($D=0$) 还是 M ($D=1$)。对于处在状态 O 或 M 的备份块,由主指针指向的 AM 是该块的拥有者,并且在发生失效时提供块的拷贝。

现在来看平坦 COMA 是怎样处理内存块的定位以及重定位的。在平坦 COMA 中,最多通过三次网络跨越可以定位到一个块,步骤如下:首先向主目录发送 BusRd 请求,主目录将请求传递给主指针指向的节点,该节点会进行响应,并且发出一个同 cc-NUMA 中处理远程失效 (图 5-21) 类似的事务流。其他一致性事务同 cc-NUMA 的 cache 目录协议中的事务类似。

我们需要关注一下 COMA 中块的重定位问题。当一个换出块处在状态 S 时,只要通知主目录,主目录将该节点从 PFV 中抹去后就可以将它丢弃了。但是,如果块处在状态 O 或 M,该块的拥有权甚至数据本身都要传送给另一个节点。由于记录了哪些节点中该备份块的状态为 S,因此,此时主目录项可以指定一个新的拥有者。

习题

5.1 图 5-27 给出的顺序伪代码实现了利用 Gauss-Seidel 红黑排序方法求解线性方程组的算法。外层循环会一直循环到满足某个收敛条件时结束,我们稍后会进行解释。外层 while 循环的每一次迭代,会访问 N 乘 N 的矩阵 A 的所有元素。每个矩阵元素通过计算它的旧值和邻居元素 (北、东、南、西) 值的平均数得到。为了简便,假设矩阵周围会有一层值为 0 的边界元素。每个元素都会在 3~9 行和 10~16 行的两个连续嵌套 for 循环中被访问,每个嵌套 for 循环都会隔一个元素进行访问。这种策略称为红黑排序,它消除了循环依赖,很大程度上简化了对这个算法的并行化。当两个连续循环中每个元素新旧值之差相加的和小于某个阈值时,算法停止迭代。在每两个 for 循环的第 8 行和 15 行通过变量 sum_of_diff 累加差值,并且在第 17 行同阈值进行比较,当累加的和小于阈值时,算法停止。我们的任务是将这个串行算法改写为共享内存系统的并行算法。

(a) 当矩阵元素没有循环依赖时,外围 while 循环的每次迭代的两个嵌套 for 循环可以并行。但是对差值求和会带来循环依赖。利用同 5.2.1 节中矩阵乘法算法一样的方法,即将累加差值的部分放入临界区,将两个嵌套循环并行化。

(b) 第一次的并行循环必须在第二次并行循环开始前结束。另外,在检测是否满足收敛条件之前,必须累加完两个嵌套循环中所有矩阵元素的差值。像 5.2.1 节介绍的那样,可以利用障碍同步来保证这个顺序。对这个共享内存的并行算法来说,应该在哪里添加障碍同步? 如果没有障碍同步会产生什么问题?

(c) 假设矩阵含有 1024×1024 个元素。每个元素的计算需要 10ns,可以用 16 个处理器并行执行。创建线程花费 100ns,并且主线程顺序创建子线程。假设顺序执行的部分,如临界区内的运


```

1 while (!converged) {
2     sum_of_diff = 0;
3     for (i=0;i<N;i++)
4         for (j=i%2;j<N;j+=2){
5             oldA = A[i,j]
6             A[i,j] = oldA + A[i-1,j] + A[i,j+1] +
7                     A[i-1,j] + A[i,j-1];
8             sum_of_diff += abs(A[i,j] - oldA);
9         }
10    for (i=0;i<N;i++)
11        for (j=(i+1)%2;j<N;j+=2){
12            oldA = A[i,j];
13            A[i,j] = oldA + A[i-1,j] + A[i,j+1] +
14                    A[i-1,j] + A[i,j-1];
15            sum_of_diff += abs(A[i,j] - oldA);
16        }
17    if (sum_of_diff/(N*N) < TH) converged = 1;
18 }

```

图 5-27 利用 Gauss-Seidel 法求解线性方程组的顺序算法伪代码

行, 每次需要 100ns, 外围 while 循环在收敛之前需要执行 10 次, 请计算获得的加速比。

- 5.2 在探索一个新的片上多处理器系统的设计时, 很重要的一个决策是 cache 组织方式的选择。在 5.4.1 节我们讨论了共享和私有 cache 之间的权衡, 在本题中, 我们定量地探讨这个问题。假设一个片上多处理器有 4 个处理器核。我们可以选择私有的或者共享的一级 cache 组织, 这两种方式消耗相同的芯片资源。下面给出了每种 cache 组织的具体条件, 两种组织的块大小都为 16 字节。

私有 cache 方式: 每个私有 cache 有 8 个块项, 采取直接映射, cache 命中时间为 1 个周期。私有 cache 间的一致性通过 5.4.2 节介绍的简单协议维护, 执行一个侦听操作的时间是 10 个周期, 如果要从内存重新载入一个块, 需要 100 个周期。

共享 cache 方式: 共享 cache 的块项数与私有 cache 中块数的总和相同。此外, 它被划分为同处理器个数相同的多个 bank, 内存块按照轮询的方式映射到不同的 bank 上, 块地址 i 被映射到 $i \bmod B$ 的 bank 中, 其中 B 是 bank 数。处理器通过使用一个 $B \times B$ 大小的交叉开关 (见第 6 章) 来访问各个 bank。如果没有竞争, 访问一个 cache bank 需要 2 个周期, 而块从内存中载入则需要 100 个周期。

(a) 如果单个处理器连续两次顺序访问内存块 0~15, 计算两种组织结构的平均内存访问时间。哪种组织方式的访存时间最短? 最短需要多长时间?

(b) 如果每个处理器连续两次顺序访问内存块 0~15, 忽略竞争因素, 计算两种方法的平均访问时间。哪种组织方式的访存时间最短? 它需要多长时间?

(c) 给处理器编号 1, ..., 4。如果处理器 i 连续两次顺序访问 $8 \times (i-1)$ 到 $8 \times (i-1) + 7$ 的内存块, 忽略竞争因素, 计算两种方法的平均访问时间。哪种组织方式的访存时间最短? 最短需要多长时间?

(d) 假设块 0~7 初始时已经存在这两种 cache 组织中。现在假设处理器 1 修改了所有块, 处理器 2 接下来读取所有块, 忽略竞争因素。哪种组织方式的访存时间最短? 最短需要多长时间?

(e) 基于之前定量获得的结果, 在什么条件下私有 cache 的平均性能优于共享 cache? 为什么很多片上多处理器的设计喜欢采用私有的一级 cache, 而二级或三级 cache 一般采用共享的?

- 5.3 考虑一个包含 8 个处理器的共享内存多处理器系统, 每个处理器有一个私有 cache, cache 一致性通过 5.4.2 节介绍的简单协议维护。总线仲裁机制根据处理器/cache 单元的编号采用固定优先级的方式, 当编号为 i 和 j 的两个单元同时发出总线请求且 $i < j$ 时, 单元 i 会得到总线。但是, 总线仲裁机制可以“记住”上次获得总线的单元, 该单元在下一个总线周期不能获得总线。现在让每个处理器都对同一个地址发送一个写请求, 并且紧接着发送一个读请求, 每个处理器向该地址写入其

单元编号。请问每个处理器的读请求会返还什么值？

- 5.4 假设一个共享内存多处理器系统中通过私有 cache 连接到共享总线上，并用 MSI 协议来维护 cache 一致性。执行各种协议动作的时间如表 5-6 所示。读命中和写命中只需要 1 个时钟周期，而读请求需要 40 个周期，因为需要将块从存储层次的下一级中取回。总线更新请求的时间比较少，这是因为它不引入块传输，而是使其他拷贝无效。这个动作包括将请求传输到总线，并且每个 cache 需要进行一次侦听，侦听动作的时间也列在了表 5-6 中。

表 5-6 协议动作的时间以及流量参数 B 是块大小

请求类型	执行协议动作所需的时间	流量
读命中	1 个时钟周期	N/A
写命中	1 个时钟周期	N/A
读请求在下一级存储中完成	40 个时钟周期	6 字节 + B
读请求在私有 cache 中完成	20 个时钟周期	6 字节 + B
独占读请求在下一级存储中完成	40 个时钟周期	6 字节 + B
独占读请求在私有 cache 中完成	20 个时钟周期	6 字节 + B
总线更新请求	10 个时钟周期	10 字节
拥有权请求	10 个时钟周期	6 字节
侦听动作	5 个时钟周期	N/A

假定有如下的对块 X 和 Y 的读写序列：R1/X, R2/X, R3/Y, R4/X, W1/X, R2/X, R3/Y, R4/X, 其中 Ri/B 以及 Wi/B 分别代表处理器 i 或者 cache 单元 i 对块 B 的读和写操作，请针对下面的每种情况分别计算上述读写序列所需要的时间。

- (a) 假设侦听动作优先于同一个单元的处理器读/写请求，这些请求必须等到侦听操作先完成。此外，标识目录没有进行复制。
- (b) 假设对标识目录进行复制，允许输入的侦听动作和输出的处理器读写协议动作同时发生。在这种情况下，同一时刻对标识目录的并发查找只能发生在侦听动作不会改变块状态的前提下。计算每个处理器执行完这些访存请求所需要的时间。
- 5.5 假设在一个共享内存多处理器中，处理器/私有 cache 单元是通过共享的单事务总线连接。基准一致性协议是 MSI 协议，但是我们想知道增加一个独占状态让其成为 5.4.3 节那样的 MESI 协议后，性能方面会有多少提升。我们采用和 5.4 题中相同的假设和标记，分别采用 MSI 协议和 MESI 协议的情况下，我们想知道执行如下访存操作序列所需要的时间是多少。处理器访问操作序列的顺序如下：R1/X, W1/X, W1/X, R2/X, W2/X, W2/X, R3/X, W3/X, W3/X, R4/X, W4/X, W4/X。
- (a) 现在假设从状态 E 到 M 的转换不产生访问开销，协议事务的访问开销见表 5-6，请计算采用 MSI 协议和 MESI 协议下执行该访问序列分别需要多少周期？
- (b) 以字节为单位，利用表 5-6 中的数据，比较 MSI 协议和 MESI 协议产生的流量，假设 B 为 32 字节。
- 5.6 假设一个共享内存多处理器中，处理器/私有 cache 单元通过共享的单事务总线连接。基准一致性协议是 MSI 协议，但是我们想知道增加一个 O 状态让其成为 5.4.3 节那样的 MOESI 协议后，性能方面会有哪些提升。我们需要通过表 5-6 给出的参数，计算采用 MSI 协议和 MOESI 协议执行一系列访存操作的时间和流量。处理器访问操作序列的顺序如下：R1/X, W1/X, W1/X, R2/X, W2/X, W2/X, R1/X, W1/X, W1/X, R2/X, W2/X, W2/X。
- (a) 采用 MSI 协议和 MOESI 协议执行这个访问序列分别需要多少周期？假设协议事务的访问开销由表 5-6 给出。
- (b) 使用表 5-6 中的数据，并且假设 B 为 32 字节，计算 MSI 协议和 MOESI 协议产生的流量分别是多少字节。

- (c) 比较 5.5 题中得到的 MESI 协议的访问开销和流量与本题中的 MOESI 协议的访问开销和流量。请分析 MOESI 协议的优势在哪里？通过什么方式可以抵消 MOESI 相对于 MESI 的优势？
- 5.7 假设一个共享内存多处理器中，处理器/私有 cache 单元通过共享的单事务总线连接。基准一致性协议是 MESI 协议，我们想知道按照 5.4.4 节所描述的增加读抄写（read snarfing）后，会带来哪些性能上的提高。使用表 5-6 给出的数据，采用 MESI 协议加上读抄写，请计算执行以下一系列访问操作的时间和流量：R1/X, R2/X, R3/X, R4/X, W2/X, R1/X, R3/X, W3/X, R1/1X, R2/X。
- (a) 采用 MESI 协议，加与不加读抄写，计算执行访问序列分别需要的周期数，协议事务的访问开销由表 5-6 给出。
- (b) 比较 MESI 协议，加与不加读抄写产生的流量，利用表 5-6 中的数据，并且假设 B 为 32 字节。
- 5.8 假设一个共享内存多处理器中，处理器/私有 cache 单元通过共享的单事务总线连接。基准一致性协议是 MESI 协议，但我们想知道按照 5.4.4 节所述利用基于更新策略的一致性协议后，会带来哪些性能上的提高。请根据表 5-6 所给出的参数，计算采用 MESI 协议和基于更新策略的协议执行以下一系列访存操作的时间和流量：R1/X, R2/X, R3/X, R4/X, W1/X, R2/X, R3/X, W2/X, R1/X, R3/X。
- (a) 采用基于使无效化的 MESI 协议和基于更新策略的协议，计算执行访问序列需要的周期数，协议事务的访问开销由表 5-6 给出。
- (b) 比较基于使无效化的 MESI 协议和基于更新策略的协议产生的流量。利用表 5-6 中的数据，并且假设 B 为 32 字节。
- 5.9 假设一个共享内存多处理器中，处理器/私有 cache 单元通过共享的单事务总线连接。基准一致性协议是 MESI 协议，但我们想知道按照 5.4.4 节增加迁移共享检测/优化后，会带来哪些性能上的提高。请根据表 5-6 所给出的参数，计算采用 MESI 协议，加或不加迁移共享检测/优化，执行如下的一系列访问操作分别所需的时间和流量：R1/X, W1/X, R2/X, W2/X, R3/X, W3/X, R4/X, W4/X。
- (a) 采用 MESI 协议，加与不加迁移共享检测/优化，计算执行访问序列分别需要的周期数，协议事务的访问开销由表 5-6 给出。
- (b) 比较 MESI 协议，加与不加迁移共享检测/优化产生的流量。利用表 5-6 中的数据，并且假设 B 为 32 字节。
- 5.10 正如我们在 5.4.5 节看到的，稳定状态（比如，MSI 协议中的状态 M、S 和 I）是无法解决数据竞争问题的。图 5-15 向 MSI 协议中增加了瞬时状态来作为从状态 S 到 M 的中间状态，但还是会出现其他数据竞争。请增加新的瞬时状态以及对应的状态转换，保证从状态 I 到 M 的正确转换。
- 5.11 考虑由 3 个处理器/cache 单元组成的共享内存多处理器系统，通过 MSI 协议维护 cache 一致性。表 5-7 给出了 3 个处理器对同一块中不同变量（A, B, C）的访问序列。

表 5-7

	处理器 1	处理器 2	处理器 3
1	R _A		
2		R _B	
3			R _C
4	W _A		
5			R _C
6		R _A	
7	W _B		
8			R _A
9			R _B

- (a) 判断失效是冷失效、真共享失效还是假共享失效。
- (b) 以上失效中，哪个失效可以忽略而不会影响程序的正确性？
- (c) 利用表 5-6 中的参数，并假设块大小为 32 字节，计算该序列导致的必要传输所占的比例。
- 5.12 考虑由 3 个处理器/cache 单元组成的共享内存多处理器系统，通过 MSI 协议维护 cache 一致性，私有 cache 采取直接映射。表 5-8 给出了 3 个处理器对 4 个变量（A、B、C 和 D）的访问序列，其中 A、B 和 C 属于同一块，而 D 属于另一个块。两个块被映射到 cache 的同一项，并且 cache 初始时是满的。

表 5-8

	处理器 1	处理器 2	处理器 3
1	R_A		
2		R_B	
3			R_C
4	W_A		
5			R_D
6		R_B	
7	W_B		
8			R_C
9		R_B	

- (a) 判断失效是冷失效、替换失效、真共享失效还是假共享失效。
- (b) 以上失效中，哪个失效可以忽略而不会影响程序的正确性？
- 5.13 在 5.4.7 节中，我们介绍了旁路转换缓冲（TLB）一致性的重要概念。当虚实页转换的映射发生改变时，我们想计算在多处理器系统中维护所有 TLB 一致性所需要的时间。假设有 4 个 TLB 包含某个页的转换信息。此外，假设调用缺页异常处理程序需要 1000 个周期，处理器间发送中断需要 100 个周期，在单个处理器上通过软件执行一个 TLB 项 shutdown 操作需要 200 个周期，无效掉某个物理页在每个 cache 中包含的块需要 20 个周期，每个处理器在 TLB 项和 cache 中无效块信息都被移除后，需要给执行缺页异常处理程序的处理器发送确认信息，这个过程需要 100 个周期。请根据上述参数计算执行 TLB shutdown 操作所需要的时间。
- 5.14 考虑一种共享内存多处理器系统的可扩展实现，系统包含多个节点，每个节点包括一个处理器、一个私有 cache 以及一部分内存，如图 5-19 所示。通过目录协议维护 cache 一致性，目录对每个内存块都维护一个存在标志向量，来记录哪些节点有该块的备份，如图 5-20 所示。在主节点和远程节点处理目录请求的时间为 50 个周期。此外，表 5-9 给出了所有一致性引起的请求和响应所需的延时以及流量，块大小为 32 字节。

表 5-9 协议动作的耗时与流量 B 是块大小

请求类型	执行协议动作所需的时间	流量
读命中	1 个时钟周期	N/A
写命中	1 个时钟周期	N/A
BusRd	20 个时钟周期	6 字节
RemRd	20 个时钟周期	6 字节
RdAck	40 个时钟周期	6 字节
Flush	100 个时钟周期	6 字节 + B
InvRq	20 个时钟周期	6 字节
InvAck	20 个时钟周期	6 字节
UpgrAck	20 个时钟周期	6 字节

- (a) 当主节点与请求节点是同一个节点, 并且内存备份是干净的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (b) 当主节点与请求节点是同一个节点, 并且内存备份是脏的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (c) 当主节点与请求节点不是同一个节点, 并且内存备份是脏的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (d) 当主节点与请求节点不是同一个节点, 远程节点与主节点是同一个节点, 并且内存备份是脏的时, 计算处理的 cache 失效的周期数, 并计算该一致性事务产生的流量 (以字节为单位)。
 - (e) 当主节点与请求节点不是同一个节点, 远程节点与主节点也不是同一节点 (当然, 远程节点与请求节点也不是同一节点), 并且内存备份是脏的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
- 5.15 采用图 5-21 中的基于目录策略的协议, 但是用表 5-9 中给出的延迟和流量参数, 并且假设块大小为 32 字节。
- (a) 当主节点与请求节点是同一个节点, 并且内存备份是干净的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (b) 当主节点与请求节点是同一个节点, 并且内存备份是脏的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (c) 当主节点与请求节点不是同一个节点, 并且内存备份是干净的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (d) 当主节点与请求节点不是同一个节点, 远程节点与主节点是同一个节点, 并且内存备份是脏的时, 计算处理 cache 失效的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (e) 当主节点与请求节点不是同一个节点, 远程节点与主节点也不是同一节点 (显然, 远程节点与请求节点也不是同一节点), 并且内存备份是脏的, 计算处理 cache 失效需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (f) 在哪种情况下延迟与流量同 5.14 题不同?
- 5.16 考虑一种共享内存多处理器系统的可扩展实现, 系统有若干节点, 每个节点包括一个处理器、一个私有 cache 以及一部分内存, 如图 5-19 所示。系统通过目录协议维护 cache 一致性, 目录对每个内存块都维护一个存在标志向量, 用于记录哪些节点有该块的备份, 如图 5-20 所示。在主节点和远程节点处理目录请求的时间为 50 个周期。此外, 表 5-9 给出了所有一致性引起的请求和响应的延时以及流量, 块大小为 32 字节。
- (a) 当主节点与请求节点是同一个节点, 并且内存中备份是唯一的, 计算写一个处在共享态的块需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (b) 当主节点与请求节点是同一个节点, 并且块的共享数为 4 时, 计算写一个处在共享态的块需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (c) 当主节点与请求节点不是同一个节点, 并且内存中备份是唯一的, 计算写一个处在共享态的块需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (d) 当主节点与请求节点不是同一个节点, 远程节点与主节点是同一个节点, 并且内存中的块备份是脏的, 计算写一个处在无效态的块需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
 - (e) 当主节点与请求节点不是同一个节点, 远程节点与主节点也不是同一个节点, 并且内存中的块备份是脏的, 计算写一个处在无效态的块需要的周期数, 并且计算该一致性事务产生的流量 (以字节为单位)。
- 5.17 在针对一个包括 128 个处理器的片上多处理器系统进行设计探索时, 架构设计团队根据图 5-23 采取分层组织方式。每个集群由 8 个处理器组成, 每个处理器有自己的私有 cache, 并且每个集群有一个共享 cache 作为二级 cache。考虑采用下列多种方法维护集群内和集群间一致性, 假定块大小

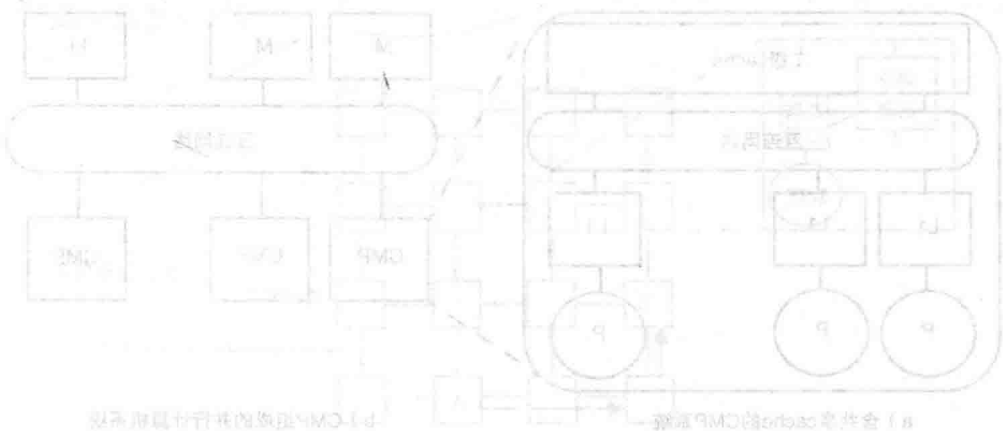
为 32 字节, 请计算维护二级 cache 和存储一致性所需要的目录信息的开销:

- (a) 每个集群内部和集群间都采用基于存在标志目录协议。
- (b) 每个集群内采用有两个指针的有限指针目录协议, 并且在集群间利用粗向量目录协议将 4 个集群分为一组。
- (c) 集群内部采用如图 5-22 所示的以 cache 为中心的目录协议, 集群间采用有 4 个指针的有限指针协议。

5.18 考虑如图 5-22 的以 cache 为中心的目录协议。如果有 N 个拷贝, 请计算执行一次使无效化的时间, 假设一个请求和一个响应需要 K 个周期。在同样的假设下, 如果采用存在标志向量协议, 花费的周期又是多少?

5.19 考虑矩阵加法 ($A = B + C$) 的并行实现, 矩阵大小为 1024×1024 , 每个元素占 8 字节。页大小是 4K 字节。共享内存多处理器系统有 16 个节点, 采用 cc-NUMA 组织方式, 每个节点包含一个处理器、一个私有 cache 以及一部分内存, cache 块大小为 32 字节。矩阵在内存中按行优先存储 (连续的行在内存空间中是连续存放的), A, B, C 矩阵是逐个连续存放的。并行算法如下: 结果矩阵 A 的前 $1024/16$ 行的元素由第一个节点计算, 接下来的 $1024/16$ 行元素由第二个节点计算, 依此类推。

- (a) 假设页布局采用静态轮询算法, 请计算每个节点访问本地 cache 和远程 cache 次数的比是多少?
- (b) 假设采用页迁移策略, 迁移开销同 16 个远程访问开销一样, 当远程访问 cache 的开销超过页迁移开销的 2 倍时, 就会进行页迁移。请计算每个节点访问本地 cache 和远程 cache 次数的比是多少?



互连网络

6.1 概述

互连网络是计算机系统中的重要组成部分。设计高性能并行计算机的关键是消除串行瓶颈，因为这些串行瓶颈会在多个级别上削弱并行机制的作用。跨处理器核的指令级和线程级并行结构，需要一套可以通过深层次的 cache 高速地将指令和数据供给处理器的存储系统。在这种环境下，假定 cache 失效的处理需要花费 100 个时钟周期，那么即使只是 1% 的 cache 失效率，也会有一半的执行时间花费在从内存到处理器的指令和数据传输上。因此，为内存和 cache 之间的指令和数据传输提供尽可能短的延迟是极其重要的。

除了延迟之外，提供足够大的存储带宽也是非常重要的。如果存储带宽不足，那么内存请求的竞争会导致访存延迟加大，进而又会影响指令的执行时间和吞吐量。举个例子，假设一个非阻塞 cache 同时支持 N 个并发 miss 请求，如果 cache 和内存之间的总线每 T 时钟周期可以传输一个 cache 块，那么处理 N 个 cache 失效需要花费 $N \times T$ 个时钟周期，而如果总线可以并行地传输 N 个 cache 块，那么处理时间将降低到 T 个时钟周期。

通常情况下，互连网络的作用是在计算机各个组件之间传输信息，特殊情况也包括在存储器和处理器之间传输数据。对于现在的并行计算机，不管是单芯片的 CMP 系统还是多机构成的大规模并行计算机系统，互连网络都是非常重要的，在这些系统中，数据从内存传输到处理器的过程中如何实现低延迟和高带宽始终是一个非常关键的问题。

图 6-1 显示了两个并行计算机系统。在图 6-1a 中，一个单芯片多处理器（CMP）上包含多个处理器核，处理器核通过私有的一级 cache 连接到一个共享的二级 cache。为了给处理器核提供可扩展的访存带宽，共享 cache 分为多个体（bank）组织，每个 bank 通过端口连接到互连网络。为了避免传输瓶颈，互连网络的带宽必须和共享 cache 的访问带宽相匹配。在图 6-1b 中，每个 CMP 作为更大计算机系统的一个计算节点，再通过互连网络把各个节点连接到内存模块。和前面的共享 cache 设计类似，为了给这些 CMP 节点提供足够的带宽，内存也划分出多个模块来组织，而且互连网络的带宽也同样需要和内存所提供的访问带宽相匹配。

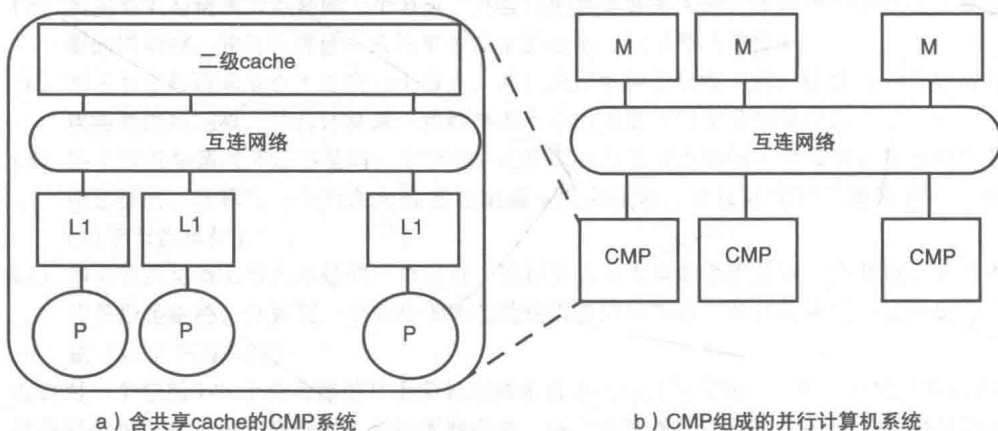


图 6-1 多处理器系统中的片上和系统互连网络

图 6-1a 所示的网络结构通常叫作片上网络（Network on Chip，NoC 或者 On-Chip Network，OCN），而图 6-1b 所示的通常称为系统区域网络（System Area Network，SAN）通常用它来将几个处理器芯片或者板级计算系统互连形成大规模的并行计算机系统。互连网络中的一种极端例子是全互连网络，比如，在 OCN 中，通过直接的点对点连接将每个一级 cache 和二级 cache 的每个端口互连，就形成了一个全互连网络，这种网络可以为 OCN 提供最大可能的访存带宽。不过这种设计是有代价的，它需要消耗大量的片上资源，最终可能导致不得不缩小一级 cache、二级 cache 的容量，并且减少处理器核的数量。因此，可以看到，片上资源的大小在很大程度上限制了互连网络的延迟和带宽。这一结论同样适用于 SAN 场合，因此，全互连网络是极其昂贵的，如果从成本的角度来考虑，这是一种不具备可扩展性的互连结构。

另一个重要的限制是功耗。CMP 系统上通常都有一定的功耗限制，这会制约片上结构的设计，包括互连网络。因此，互连网络的设计目标是通过一定的成本（比如硅面积）和功耗限制实现尽可能的低延迟和高带宽。

在这一章中，我们将不会探讨将多个计算机通过局域网（LAN）或者广域网（WAN）互连的问题。虽然它所面对的设计考虑是相似的，但是权衡考虑的标准和实际可行的方法之间还是有很大的差异，因为在这类系统中，延迟往往不是最关键的考虑因素。

本章的主要内容如下：

- 6.2 节主要介绍互连网络的基本设计概念，互连网络的功能和结构，拓扑结构的作用，交换策略以及路由算法。
- 6.3 节主要介绍交换策略以及它们对延迟和带宽的影响。本节将详细介绍 4 种不同的交换策略：电路交换，存储转发，直通式，虫洞交换。
- 6.4 节主要介绍不同的互连网络拓扑结构以及各自的延迟和带宽特性。
- 6.5 节主要介绍不同拓扑结构的路由算法，包括死锁避免技术和自适应路由算法。
- 6.6 节主要介绍不同缓冲区选择下的互连网络交换结构。

6.2 互连网络的设计空间

互连网络的作用是将一定数量的计算节点互相连接起来，使得这些不同资源节点之间可以互相传输信息，网络中的节点可能是 cache 模块、存储模块、计算机主板或者是片上多处理器。

6.2.1 设计概念综述

首先介绍一些互连网络设计中的基本概念。先从一个具体的互连结构——二维网络结构（mesh）出发，如图 6-2 所示。

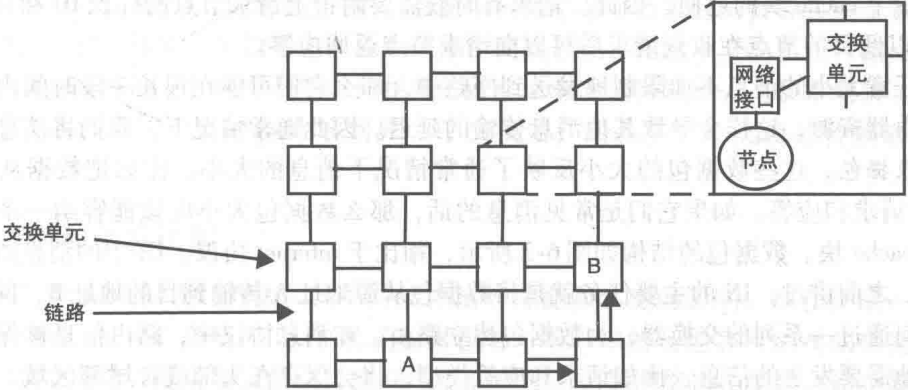


图 6-2 连接 16 个节点的 mesh 互连网络

图中网格将 16 个交换单元（也称交换器或交换机）以 4×4 的拓扑结构连接起来，每一个节点通过一个网络接口和交换器连接，如图 6-2 的右上部分所示。交换器通过双向的链路和它周围东、西、南、北四个方向上的邻居连接起来。交换器通常有多个输入和输出，它最简单的作用就是将一个输入和输出连接起来。网络中的链路本质上是一些电缆或者光纤，用于在一个交换器的输出端口和另一个交换器的输入端口之间传输数字信息。

链路

链路主要用来连接不同交换机和网络接口的输出端口和输入端口，链路的宽度是在一个时钟周期中可以并行传输的比特位数。因此链路的宽度和一个时钟周期的长度决定了链路的带宽：假如宽度为 w ，一个时钟周期的长度为 t ，那么带宽为 w/t 比特每时间单元。链路的宽度主要受 OCN 占用的芯片面积或者 SAN 中的电缆布线费用的限制，而链路的时钟周期长度方面，需要区分是端口间的同步还是异步传输。在同步传输模式下，链路和交换器使用相同的时钟，其时钟周期长度是由最慢的组件（比如交换器）决定的。与之相反，在异步通信模式下，不同的组件可以有不同的时钟周期，通过握手协议来进行同步。

交换单元

交换单元（switch element，简称 switch）的作用是在输入端口和输出端口之间建立连接，并且在连接建立之后负责两个端口之间的信息传输，交换器一般具有确定数量的输入和输出端口，且二者数量一般相同。带有 n 个输入和 n 个输出的交换器一般称为 $n \times n$ 交换器，其交换度数（switch degree）为 n 。因为多个输入端口可能都需要往同一个输出端口传输数据，因此可能会出现针对某个特定输入端口的竞争情况。为了解决竞争问题，交换器通常具有缓存信息的能力，可能实现在输入端口，也可能实现在输出端口，或者两端都能缓存。图 6-2 中，不考虑连接自身节点的输入和输出端口，这个 4×4 网格就是由度数是 4 的交换器组成的。输入端口和它们的缓冲区通常是通过一个交叉开关连接到输出端口和对应的缓冲区。在后面的 6.4 节中，我们讨论到网络的拓扑结构的时候会再介绍交叉开关的设计。

基本的互连网络功能

从源节点传输到目的节点的单元信息称为消息，消息的大小可能从单字长到任意字长不等。单字长消息的典型例子是共享内存的系统中写穿透 cache 发送的更新内存的请求。在一些并行的程序模块中，比如第 5 章讨论的信息传递机制中，节点之间可以交换任意大小的消息。

在较高级别的协议层或者并行程序模块之间的传输需求可能需要不同类型的信息传输机制，最简单的情况是请求节点向单个目的节点发送消息，即单播请求，此外，请求节点向一组目的节点发送消息的情形称为多播请求，而如果是向所有节点发送消息的话，则称为广播请求。在这些消息请求中，有些是需要应答的，比如从 cache 模块向内存模块发送一个读请求，会期望有一个 cache 块的返回。因此，请求有时候需要附上请求节点的标识 ID 和目的节点的标识 ID，以便目的节点在收到请求后可以向请求节点返回应答。

如果任意大小的消息不加限制地发送到网路中，那么它们可能在很长一段时间内都会完全占用掉路由器资源，这样会导致其他消息传输的延迟。因此通常情况下，我们将消息分解成固定大小的数据包，这些数据包的大小反映了通常情况下消息的大小。比如把数据从内存取到 cache 层的请求和应答，如果它们是常见消息的话，那么数据包大小应该能容纳一个简单请求或者一个 cache 块。数据包的结构如图 6-3 所示，相比于 internet 协议，IN 中的消息协议大体上比较简单。之前讲过，IN 的主要任务就是将数据包从源地址 A 传输到目的地址 B，网络至少需要知道如何通过一系列的交换器，为数据包建立路由。在消息协议中，路由信息被保存在头部区域，其他需要发送的信息，比如请求和应答类型，将会保存在头部或者尾部区域。由于辐射可能造成的软错误或者永久性交换/链路失效造成的硬错误等原因，数据包可能出现内容丢失

或者被篡改的情况，所以还需要在 IN 协议级别或者更高层次级别实施预防错误的保护机制。假如在互连层级别来实施保护，数据包必须包含错误代码区域，用于错误的检测和修正。除了上述内容之外，剩下的需要在源和目的节点进行传输的信息（比如在 cache 块中的数据）不是互连网络实现自身功能所关注的内容，互连网络只需将数据包从源节点传输到目的节点。所有未被互连网络解析的信息将被保存在数据包的净负荷区域，这些内容是上层协议关心的。暴露给 IN 的信息一般称为数据包信封，它包含头部信息、错误代码信息和尾部信息。

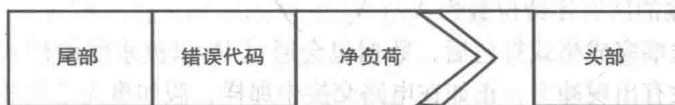


图 6-3 数据包构成

接下来我们开始描述在消息从图 6-2 中网格的节点 A 传输到节点 B 的过程中到底发生了什么，并且提出一些设计问题供后续讨论。被传输消息首先被分割成数据包序列，这个分割是由更高层协议完成的，并且网络接口通过一系列软硬件结合机制提供支持，这些数据包先被保存在缓存区，随后逐个被注入 IN 中。

第一个需要考虑的设计问题是决定数据包是如何在 IN 中路由的，这由交换策略决定。最简单的策略是只要有数据包从节点 A 传输到节点 B，就通过网络建立一个路由，这种方式称为电路交换策略，这种策略的优点是数据包从发送方传输到接收方的时候不会被干扰。在电路交换策略中，数据包不需要携带路由信息，因为在数据包传送之前，路由已经建立好。

电路交换策略避免了在每次交换时的路由开销，缺点是在整个数据包传输到目的节点之前，其他的数据包无法获得该通信路径上的任何网络资源，这可能会降低其他数据包传输的可用带宽。而且数据包越大，在同一个路由上等待资源的其他数据包就会越多。

另一个方式是数据包交换。在数据包交换策略中，数据包从一个交换器传输到另一个交换器，并且只在需要的时候占据一个交换器资源。在本章的后续内容中，我们将详细回顾数据包交换策略的不同选项，以及这些选项给延迟和带宽带来的影响。

和交换算法密切相关的一个问题是路由算法。假设数据包交换以及路由决策功能都是分布在各个交换器上的。在图 6-2 中，被标记的路由首先经过 X 轴（水平方向）的正向，然后经过 Y 轴（垂直方向）的正向传输数据包，这种路由算法称为维度优先路由，它受网络拓扑结构的影响（在这个例子中，是网格拓扑结构）。拓扑结构首先会影响 IN 的延迟、带宽和成本等。在 6.4 节中，我们会详细地介绍一些网络拓扑结构和它们的路由算法。

数据包交换策略使得多个数据包传输可以同时进行，它们可以根据路由来共享一些相同的链路和交换器资源。之所以能做到这一点，是由于分配给某个数据包的资源在整个数据包离开交换器时会被回收。如果一个数据包被分配到某个交换器上，而这个交换器正在路由另一个数据包的话，就会出现冲突，这时需要设计相应的策略来解决这种冲突，这种策略称为流控，它决定何时将数据包从一个交换器传输到另一个交换器，或者从一个交换器传输到一个网络接口。

6.2.2 延迟和带宽模型

为了研究 IN 的不同设计方法以及不同的方法对性能和带宽的影响，现在介绍一个简单而实用的分析模型，它可以分析在 IN 中发送数据包时，端对端的延迟和所消耗的带宽。我们先从最简单的模型开始，随着覆盖的设计空间维度的增加，比如网络拓扑结构、路由算法、交换策略流控制策略等，这个模型会变得更加复杂。

端对端的数据包延迟模型

考虑一个数据包从发送节点到接收节点的传输, 假如数据包在源节点的缓冲区中已经就绪等待传输, 那么端对端的延迟就是从现在开始直到整个数据包通过 IN 传输完成, 并且在接收节点的缓冲区接收到这个数据包。发送端节点的数据包需要准备好, 数据包的头部、尾部以及可能的错误代码会被加到有效载荷区中, 如图 6-3 所示那样。假如数据包的有效载荷包括 N_p 个比特位, 而头部、尾部和错误代码所需的比特位数为总共为 N_E (这里的 E 代表信封, Envelope), 那么需要传输的所有比特位数为 $N_p + N_E$ 。

当数据包在发送端完成格式打包后, 数据包会通过 IN 以流水线的形式发送。先假设在所有传输的数据包中没有出现冲突, 正如在电路交换中那样, 假如事先已经建立好路由, 并且假定链路在一个周期可以传输一个比特位, 那么将整个数据包注入 IN 中需要 $N_p + N_E$ 个时钟周期。图 6-4a 显示了这个传输流水线的结构, 图 6-4b 表明了在没有其他数据包传输干扰的情况下, 传输过程中不同的延迟组成部分, 所有延迟组成的总和称为卸载端对端数据包延迟。

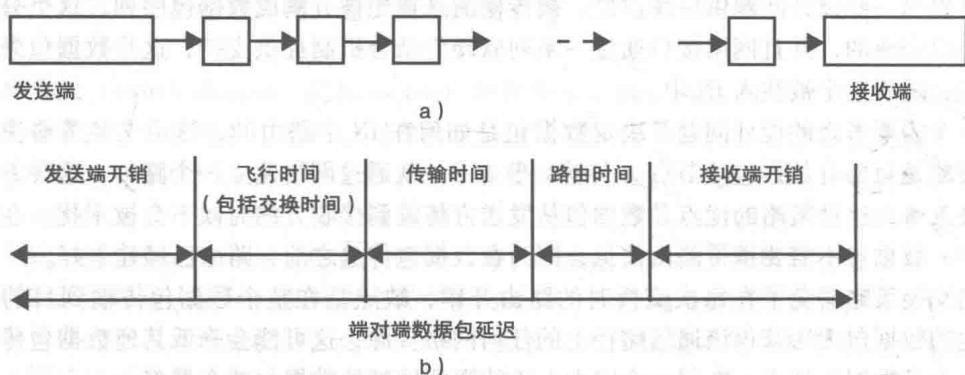


图 6-4 从源节点发送数据包到目的节点的端对端延迟模型

下面详细分析延迟的不同组成部分。

- **发送端开销。**这部分延迟指的是发送端数据包的准备工作, 包括数据包信封的生成, 以及将数据包注入网络接口缓冲区。这部分延迟通常比较固定, 随着数据包的增大, 它对端对端整体延迟的影响会相应变小。
- **飞行时间。**这部分时间是指从发射端发送一个比特到接收端所需要的时间下界。在电路交换策略中, 路由事先已经建立好, 所以单个比特可以流水的形式通过路由上的链路和交换机到达接收端, 飞行的时间由图 6-4a 中的逻辑传输流水线的长度所决定, 它取决于多个因素, 包括网络的拓扑结构、链路的数量、数据包需要经过的交换器的个数, 以及交换器的周期执行时间和复杂性等。而在数据包交换策略中, 还会受到交换时间的影响, 比如交换策略所需开销的不同也会影响最终的飞行时间。
- **传输时间。**这部分延迟是指当第一个比特数据到达接收端开始, 到所有的比特数据都已经从发送端传输到接收端的时间, 它取决于链路的带宽。通过链路可以在一个时钟周期内传输的数据块通常称为物理传输单元, 或者简称为 phit。假如 phit 的大小为 N_{ph} , 一个时钟周期的长度为 $1/f$, 那么链路的带宽为 $N_{ph} \times f$, 而数据包的传输时间就是数据包的位数除以链路的带宽, 即 $(N_p + N_E) / (N_{ph} \times f)$ 。
- **路由时间。**在电路交换策略中, 这个延迟是指在开始传输数据包之前的路由建立时间, 而在数据包交换策略中, 这个延迟是指为每个路由器设置路由策略的时间。
- **交换时间。**交换策略表明数据包是如何在交换机中传输的。数据包交换的一种实现方式是存储转发 (store-and-forward) 交换, 即将数据包先整体从一个交换机传输到另一

个交换器，然后再决定下一步路由。另一种实现方式是直通式（cut-through）交换，它以流水的形式在交换器之间传输数据包。交换策略引起的时间开销其实已经在飞行时间中计算过了。

- **接收端开销。**发送端需要创建数据包信封，与之相对应，接收端就需要打开信封将数据包从输入缓冲区取出，以便后续的数据包还可以被继续接收。这个过程引起的时间开销在接收端开销中统计。

这样，端对端的延迟就是图 6-4b 中 5 部分延迟开销的总和：

$$\begin{aligned} \text{端对端的数据包延迟} = & \text{发送端开销} + \text{飞行时间} + \text{传输时间} \\ & + \text{路由时间} + \text{接收端开销} \end{aligned} \quad (6.1)$$

在上面的式子中，交换时间被包含在飞行时间中了。

下面给出一个使用式（6.1）来计算数据包传输的端对端延迟的示例。

例 6.1 一个 100bit 数据包（包含数据包信封）在电路交换网络中传输，在路由上一共有 9 条链路。Phit 大小为 10bit，时钟周期长度为 10ns。假定在发送端和接收端的时间开销都为 10ns，并且假设建立路由的时间为 200ns，那么传输数据包所需的端对端延迟是多少？

从上面的假定，可以获取如下延迟：

- 发送端开销 = 10ns
- 路由时间 = 200ns
- 接收端开销 = 10ns

路由时间是预先建立路由的时间开销，要完成路由建立，发送端先送出一个包含单个 phit 的数据包，该数据包完整经过 9 条链路后到达接收端，然后接收端返回一个数据包来确认路由已经建立。

至于飞行时间，因为传输的长度是九级流水，每个时钟周期的长度为 10ns，因此总的飞行时间为 90ns。总共需要传输的数据量是 $100 \times 8 = 800\text{bit}$ ，phit 的大小为 10bit，因此数据包传输需要 80 个时钟周期，总的传输时间为 800ns。电路交换策略下没有交换的开销，因此交换时间为 0。通过式（6.1），我们可以得出端对端的延迟为：

$$10\text{ns} + 90\text{ns} + 800\text{ns} + 200\text{ns} + 10\text{ns} = 1110\text{ns}$$

这个例子显示，数据包经过的链路和交换器的带宽以及数量，对端对端的延迟有很大的影响，特别是对于大的数据包影响更大，而链路和交换器的数量一般由网络的拓扑结构所决定。

网络直径的概念定义了给定网络拓扑结构中传输流水级的数量，网络直径是 IN 中任意两个节点中最长的传输流水级。以图 6-2 中 4×4 的网格 IN 为例，通过统计数据包要经过的链路数（又叫路由距离）可以看到，拥有最长传输流水级的是从最左下角的节点到最右上角节点之间的一条路径，这种情况下，数据包一个经过了 6 条链路，因此，网络直径是 6。如果是其他的 $N \times N$ 的网络的话，这个网络直径就是 $2(N-1)$ 。网络直径反映的是在最差情况下的传输流水级长度，或者说是网络中两个节点在最差情况下的距离。除了网络直径，另一个常用的度量标准是平均路由距离，它反映了传输流水级的平均长度或者说是 IN 中所有节点对的平均距离。

到现在为止，我们考虑的是发送单个数据包的端对端延迟，而当消息大小超过最大的数据包负载的时候，就必须在发送端将消息拆分成多个数据包。一个有趣的问题是背靠背连续传送多个数据包的时候，端对端延迟是多少？为了传输多个连续的数据包，首先要在发送端组装数据包，这占用了发送端的一部分开销，然后把数据包注入 IN 中。假设我们采用数据包交换策略，那么当传输时间结束之后，当前数据包的最后一个比特数据离开网络时，发送端的网络接

口又可以重新注入下一个数据包了，因此，可以将组装数据包和注入网络这两个任务看成逻辑流水级的两个阶段。假如组装数据包比将数据包注入网络中所花费的时间少，那么发送端的开销就可以被隐藏起来，同样，对于接收端也是类似。实际上，发送端、互连网络和接收端组成了一个三级的流水线，最慢的那一级决定了流水线的整体吞吐量。因此，发送同一个消息对应的 N 个连续数据包的端对端延迟的计算公式如下：

$$\begin{aligned} \text{端对端延迟} = & \text{发送端开销} + \text{接收端开销} + \text{飞行时间} + \text{传输时间} + \text{路由时间} \\ & + (N - 1) \times (\max(\text{发送端开销}, \text{传输时间}, \text{接收端开销})) \end{aligned} \quad (6.2)$$

虽然交换时间可以单独计算，但是在本书中，我们将它合并到飞行时间中，因为它影响了传递数据包第一个比特的时间。另外，在数据包交换策略下，路由时间其实也是类似的，但是我们选择将它作为公式中一个独立计算项。

例 6.2 计算发送 10 个数据包的端对端延迟。假定传输时间为 100ns，发送端和接收端的开销分别是 110ns 和 80ns，飞行时间为 20ns，其中包括交换时间。路由时间为 0。

因为发送端的开销比传输时间稍长了点，所以发送端将数据注入 IN 中的速度要慢于 IN 消耗数据包的速率。相反，由于接收端的时间开销比传输时间要小，那么接收端可以按照 IN 的速率来接收数据包。综上所述，我们可以计算得到传输 10 个数据包的端对端延迟为：

$$110 + 80 + 20 + 100 + 9 \times 110 = 1300\text{ns}$$

带宽模型

带宽和延迟并不是完全独立的，带宽受限的话通常也会导致延迟变长，因此，对 IN 中带宽特征的理解就显得非常重要。首先，弄清楚在点对点的传输中，发送端到接收端的路径上是否存在带宽瓶颈，这是非常重要的。其次，即使带宽不存在瓶颈，如果与其他传输节点对中的数据包的冲突，也可能导致传输滞后，这也就意味着会导致更长的延迟。

当多个数据包在一个节点对传输的时候，如果发送端和接收端的时间开销比传输时间要小，那么这两者的开销可以被隐藏掉，也就是说，如果传输时间更长，那么可用的带宽受限于 IN 而不受限于节点。与之相反的是，如果发送端或者接收端的时间开销比传输时间长，那么带宽将不再受限于 IN，而是受限于数据包注入网络或者从网络接收的速率。因此，在节点之间传输数据包序列的有效可用带宽可以按照如下公式计算：

$$\text{有效带宽} = \frac{\text{数据包大小}}{\text{Max}(\text{发送端开销}, \text{接收端开销}, \text{传输时间})} \quad (6.3)$$

式 (6.3) 说明带宽受限于 3 个可能的瓶颈：发送端，IN，接收端。

即使在两个节点间的传输流水级没有任何带宽瓶颈，多个路由之间共享的 IN 资源也可能造成网络竞争。比如在图 6-2 的 4×4 网络中，交换器在同一个时钟周期内最多可能接收到 5 个数据包（包括和交换器相连的那个节点发来的数据包），当出现这种情况时，交换器需要能够选出一个优胜者进行处理，而阻塞其余的 4 个数据包，这个选出优胜者的过程叫作仲裁。如何处理那些被阻塞的数据包是由流控策略所决定的，但是不管使用什么策略，传输滞后就意味着会有更长的数据包延迟。这一影响被包含在端对端的延迟模型中，作为延迟的一个额外组成部分，我们称之为流控时间。

网络竞争不仅影响了端对端的延迟，也影响了有效带宽，我们之后会探究不同的流控策略以及它们对端对端延迟和带宽效率的影响。当网络竞争超过特定的阈值时，网络就会饱和，延迟也会大幅度增加，这和交通堵塞后高速公路上发生的情况类似：在拥堵的地方会逐渐开始排队，并最终导致该地段的交通完全堵塞。不过幸运的是，互连网络不是孤立地起作用，排队的数据包序列可能一直堵塞到发送节点，这使得发送节点会停止向网络中注入数据包。由此可见，IN 和节点事实上组成了一个自我调节的闭合回路，网络竞争的加剧往往会导致注入速率

的降低,而注入速率的降低又会反过来缓解网络竞争。这和交通堵塞的情况在收音机里广播之后,高速公路上发生的情况相似:当收到交通堵塞消息时,汽车会避免驶向阻塞的高速公路路段,从而降低交通的压力,帮助解决阻塞问题。

我们有必要研究一下 IN 能够提供的聚合带宽,度量标准之一是使用对分带宽。IN 可以被看成边和顶点构成的图,在图中顶点代表交换器,边代表链路。对分指的是将图按最小数量的边进行切分,使得切分后形成两个同构的子图。对分带宽指的是两个子图之间的总带宽。

例 6.3 假如在图 6-2 网格中的每个链路的带宽为 b ,那么网格的对分带宽是多少?

我们既可以将 4×4 的网格水平切割成 2×4 网格,也可以垂直切割成 4×2 网格。在任何一种情况下,切割都需要跨越 4 条链路,所以对分带宽为 $4b$ 。

当某半个网络中的节点只和剩下的半个网络中的节点通信时,对分带宽可以用来测量其网络带宽。对于并行机器中的通信模式来说,这是一个很悲观的假设。因为实际上,可扩展并行算法中的通信一般可能更具有局部性,而如果过分聚焦在对分带宽上可能导致网络设计过度。

IN 聚合带宽的另一个度量标准是节点划分出来的所有链路的带宽总和。假定在 $N \times N$ 的网格中链路的带宽是 b ,那么聚合带宽是 $2N(N-1)b$,每个节点的带宽是 $2(N-1)b/N$,因此,当 N 很大时,这个值趋近于 $2b$ 。不管我们采用的度量标准是对分带宽还是每个节点的链路带宽,网络的拓扑结构都会对 IN 带宽产生很大的影响。另外,具体的实现方法,比如每个链路的宽度、交换器忙于路由策略的时间等,都会对带宽和网络竞争产生巨大的影响。

下面我们将详细探究 IN 设计的空间,包括交换策略、网络的拓扑结构以及路由算法。

6.3 交换策略

交换策略关系到路由器是如何建立的,以及数据包是如何从源节点传输到目的节点的。为了对不同的交换策略进行性能比较,我们采用类似 6.2.2 节中的方法对源和目的之间的路径进行传输流水级的建模。进一步,假设数据包在从源节点到目的节点总共需经过 L 个交换器,每个数据包包含 N 个 phit,每个 phit 指的是传输流水级之间在单个时钟周期内能够完成的传输数据。

首先考虑电路交换策略,在电路交换策略中,首先需要建立好路由,然后数据包通过网络进行流水传输。根据 6.2.2 节中描述的端对端数据包延迟模型,路由时间是在数据包传输之前建立路由所消耗的时间。假如在每个交换器中的路由决策需要 R 个网络周期,路由时间可以表示为从源节点发送单个 phit 数据包到目的节点的时间,再加上返回到源节点通知路由已经建立的时间,因此路由时间可以表示为

$$\text{路由时间} = L \times R + \text{飞行时间} = L \times R + L = L(R+1) \quad (6.4)$$

所以端对端的数据包延迟可以表示为

$$\begin{aligned} \text{端对端的数据包延迟} &= \text{发送端开销} + \text{飞行时间} + \text{传输时间} + \text{路由时间} + \text{接收端开销} \\ &= \text{发送端开销} + L + N + L(R+1) + \text{接收端开销} \\ &= \text{发送端开销} + L(R+2) + N + \text{接收端开销} \end{aligned} \quad (6.5)$$

式中, L 和 N 分别表示飞行时间和传输时间; $L(R+1)$ 表示由式 (6.4) 得出的路由时间。如果路由算法比较简单, R 应该是一个很小的整数。

式 (6.5) 表明假如数据包比较大,传输时间将主导端对端的延迟。这种情况下,电路交换策略将会更有优势,因为数据包可以全速发送,在数据包传输过程中没有任何交换或者路由的开销。需要注意的是,这个端对端的延迟模型中没有考虑到 IN 的资源在整个数据包传输过程中会一直被占用,这实际上会对带宽利用率产生严重影响,而在上面的端对端延迟模型中并

没有考虑这个因素, 如果考虑进去的话, 可能会导致电路交换策略不像现在这么有吸引力。

在数据包交换策略中, 当数据包从源节点传输到目的节点的过程中, 路由的建立是通过在数据包到达的每个交换器中执行路由决策来完成的。下面考虑两种交换策略: 存储转发交换策略, 直通式交换策略。在存储转发交换策略中, 数据包是以整体的形式从一个交换器传输到另一个交换器的, 这意味着数据包的所有数据都必须接收完了, 才能开始往下一个节点传输数据。图 6-5 的时序图表明了了在存储转发模式下 (图 6-5a) 和直通式模式下 (图 6-5b) 数据包是如何沿着 L 个交换器从源节点移动到目的节点的。根据前面的定义, 飞行时间包括了假定没有路由开销时从源节点传输单个比特到目的节点的时间。而在存储转发交换策略中, 在数据包开始往下一个节点传输之前, 整个数据包必须从一个交换器传输到另一个交换器, 因此, 数据包在每次交换中共需花费 N 个周期, 目的节点接收到第一个比特数据时, 一共需花费 $L \times N$ 个周期。可以看到, 和电路交换策略不同, 存储转发交换策略下的飞行时间取决于数据包的大小。传输时间和电路交换策略下一样, 需要 N 个周期。假定在每个节点上的路由开销为 R 个周期, 那么总的路由时间为 $L \times R$ 个周期。因此, 端对端的数据包延迟可以表示成:

$$\begin{aligned} \text{端对端的数据包延迟} &= \text{发送端开销} + L \times N + N + L \times R + \text{接收端开销} \\ &= \text{发送端开销} + N(L + 1) + L \times R + \text{接收端开销} \end{aligned} \quad (6.6)$$

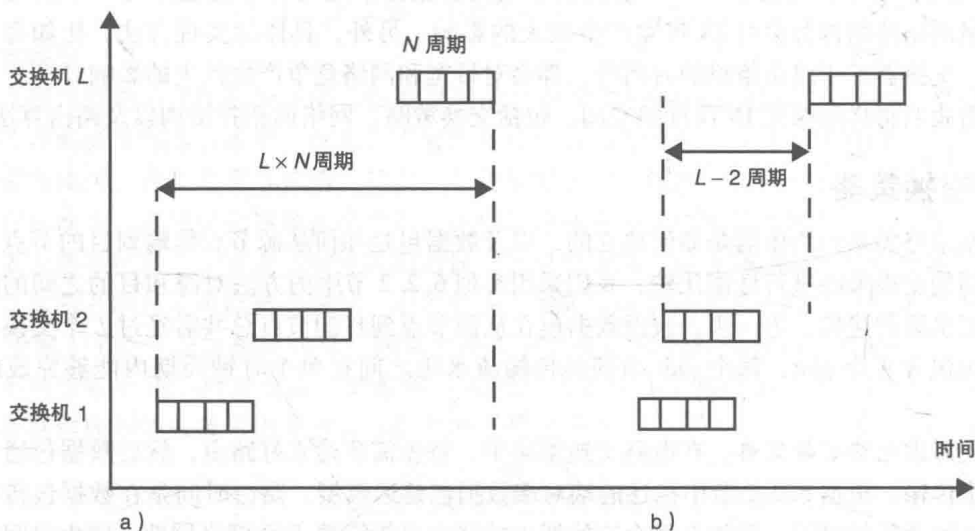


图 6-5 在存储转发 (a) 和直通式 (b) 交换策略下的数据包交换过程

因为传输时间 N 和飞行时间 $N \times L$ 是和数据包大小 N (phit 的数量) 成比例的, 所以存储转发交换策略的延迟随着数据包的增长而增长, 因此, 当数据包越大时, 存储转发模式策略的吸引力就越小。

在直通式交换策略中 (如图 6-5b 所示), 每个交换器检测带有路由地址的数据包头部。和电路交换不同的是, 这里只有当数据包到达交换器的时候, 它才会进行路由操作, 因此每次都会有 R 个周期的时间开销。只要不存在竞争, 数据包就会以流水的方式从源节点传输到目的节点。不过一旦出现竞争, 整个数据包都会被阻塞。流控单元 (flow-control unit, 简称 flit) 这个词定义的就是在竞争中被阻塞的数据包中的部分, 而在直通式交换策略中, flit 包括整个数据包的数据。

为了计算直通式交换策略的端对端延迟, 首先假设路由时间为 0, 数据包以在电路交换策略下相同的路径通过整个 IN, 因此飞行时间是 L 个周期。假如在每个交换器中路由开销为 R 个周期, 那么路由将会在端对端延迟中增加 $L \times R$ 个周期。因此, 对于直通模式的交换策略,

端对端的数据延迟可以表示为

$$\begin{aligned}\text{端对端的数据包延迟} &= \text{发送端开销} + L + N + L \times R + \text{接收端开销} \\ &= \text{发送端开销} + L \times (R + 1) + N + \text{接收端开销}\end{aligned}\quad (6.7)$$

直通式交换策略的延迟模型和电路交换策略延迟模型（见式（6.5））非常相似，但也有一个很大的区别。在电路交换策略中，在数据包从源节点传输到目的节点的过程中，整个路由是一直都设定好的。与之相反，在直通式交换策略中，数据包在交换完成时会及时地回收交换器资源，因此直通式交换策略对网络带宽的利用率要好于电路交换策略。

直通式交换策略会引起一个有趣的问题：当两个数据包到达同一个交换器并且被分配到一个输出端口的时候如何解决竞争问题。通常有两种解决方法：第一种方法是虚拟直通式（virtual cut-through）交换，在这种策略下，在交换器中的缓冲区一定要足够大，可以容纳整个数据包，当输出端口忙碌的时候，被分配到这个端口的新的数据包就被缓存到交换器的缓冲区中，因此这种情况下流控单元包括了整个数据包，这意味着当网络负载增加的时候，虚拟直通式交换策略表现得越来越像存储转发交换策略，数据包将整体从一个交换器移动到另一个交换器。另一种方法是虫洞式（wormhole）交换，由于缓冲区资源是很宝贵的，有些网络无法在交换器的输入端口提供足够的缓冲区空间来保存整个数据包。因此在虫洞交换策略中，交换器只缓冲包含路由信息的那几个 phit，因此，对应的 filit 会比数据包要小。在虫洞交换策略中，组成数据包头部的 phit 会在出现竞争的时候阻塞在当前交换器，而属于数据包的其他 phit 则缓存在阻塞节点前的其他交换器缓冲区中。可以看到，数据包被拆分保存在传输路径上的多个交换器中，这条通路将一直被保持直到数据包可以继续往前移动。

6.4 拓扑结构

我们看到端对端的数据包延迟严重依赖于数据包路径中的交换器的数量，此外，传输中由于数据包竞争造成的延迟影响在很大程度上取决于网络交换器连接的方式，因此拓扑结构在网络中扮演着很重要的角色。

在这一节，我们主要探讨通用 IN 拓扑结构的特性。在 6.4.1 节，我们先从间接互连网络这一类拓扑结构开始介绍，在这种拓扑结构中，整个 IN 可以被看作带有端口的盒子，这些端口和所有需要互相交换信息的其他节点相连。而图 6-2 所示的网络结构与此不同，网络中节点通过交换器紧密结合在一起，这种网络叫作直接互连网络，这部分内容将在 6.4.2 节进行介绍。

6.4.1 间接网络

总线

目前为止，最常见的间接 IN 拓扑结构是总线结构，总线结构也是共享媒体 IN 的一个实例。在最简单的实现中，总线连接了一系列互相之间可以通信的节点。如图 6-1a，当通过总线连接私有 cache 和共享 cache 时，可能的请求操作包括三种：第一种是读请求，该请求返回一个 cache 块；第二种是写回请求，该请求强制将脏的 cache 块写回到共享 cache 中；第三种是特殊的广播请求，这和我们第 5 章中介绍的一样，通过这种广播请求可以保证私有 cache 中内容的一致性。

因为总线是一个共享的媒体 IN，所以请求节点需要首先获取对它的独占式访问，这可以通过和总线有关的仲裁机制来实现。仲裁机制在所有请求节点中选出一个胜出者，只有胜出者的请求才允许被处理，而没有获取到访问权限的其他节点就必须等待总线释放。除了保证对总线的独占式访问，仲裁的另一个功能是保证竞争请求的公平性。

在大多数情况下,请求发出之后会跟着产生应答,比如当一个 cache 块请求被发送到内存模块,内存模块在经过一定的访问时间之后,会将请求的块响应给请求节点。从发送请求、访问内存,一直到请求块响应这个过程的时间有多长,那么其他被挂起的请求就必须等待多久。下面通过一个例子来展示一个原子的、不支持流水的总线,这种结构的带宽利用率很低。

例 6.4 假定总线结构中有三条独立的总线,在每个总线周期内,可以分别传输一个请求、一个地址以及 256bit 的数据,总线的频率为 100MHz。和总线相连的内存分体组织,可以在 200ns 访问 32 字节的 cache 块。假定这是一条原子总线,那么总线空闲的时间比例是多少?

当请求节点获取使用总线权限的时候,它在总线上提交了相应的读请求和请求块的地址。由于所有的连接节点都在监听总线的活动,因此内存模块可以直接接收来自总线的地址。这个发送请求的过程占用了 10ns 的总线,然后总线会空闲 200ns (访问内存的时间),之后再经过 10ns,内存模块将会返回本次请求的数据块和地址。因此,这个过程中总线空闲的时间比例为 $200/220 = 91\%$ 。

为了提高带宽的利用率,有必要在某个节点请求处理阶段允许其他的节点使用总线。要做到这一点,一个总线事务可以分解成请求和应答两个阶段。在流水的 (pipelined) 总线或者分割-事务 (split-transaction) 总线中,总线只在请求或者应答阶段允许独占式的访问。在例 6.4 中,当内存模块捕获了读请求之后,总线就会被释放,在内存取回数据块的过程中,其他等待的请求可以使用总线。一旦内存准备好应答,就需要重新通过仲裁获取总线,这会增加总线事务的延迟。因此,设计时往往需要在更短延迟还是更高带宽之间进行权衡,不过,通常来讲,能获取更高带宽的分割-事务总线往往更有优势。

总线结构简单,因此使用起来也很方便,但是由于总线结构属于共享媒介互连结构,本质上是不可扩展的。当连接到总线上的节点越来越多时,两个因素会导致延迟的增加和带宽的降低:首先节点的增加往往意味着总线更长;其次,节点的增多会导致总线电容负载的增加。因此,能够连接到总线上的节点数目是非常有限的,由于这个原因,诸如交叉开关等其他间接网络逐渐流行起来。

交叉开关

交叉开关是一种简单的间接网络,它通过输入/输出端口将两组节点连接起来。在图 6-6a 中,输入端口在左侧,输出端口在上侧。 $N \times N$ 的交叉开关可以看成 N 个水平的和 N 个垂直的总线,在每个相交的地方都有一个交叉点,每个交叉点可以连接一条水平总线和一条垂直总线。通过路由机制可以控制所有的交叉点,比如控制水平总线 i 可以连接到垂直总线 j (点对点或者单播连接),或者连接到多条垂直总线 (多播或者广播连接)。这意味着在所有的 N 对节点中,只要某个节点对和其他的节点对不同,那么就必须为其单独建立一个点对点的连接,可见连接数和节点数是相对应的,因此交叉开关的带宽可以随着节点数线性增加。然而节点数的增多会导致交叉开关的不断增大,随之总线的长度也随着节点数线性增加,这会导致延迟的增大。除此之外,交叉点的数量和节点数呈平方关系,交叉开关的扩展性最终会受限于可用的资源成本。

多级互连网络

为了解决交叉开关的可扩展问题,方法之一是把交叉开关作为基本结构单元来组建更大的间接网络。图 6-6b 显示了一个多级互连网络 (Multistage Interconnection Network, MIN) 的例子,用了多个 2×2 的交叉开关将左边的 8 个节点和右边的 8 个节点互连起来。

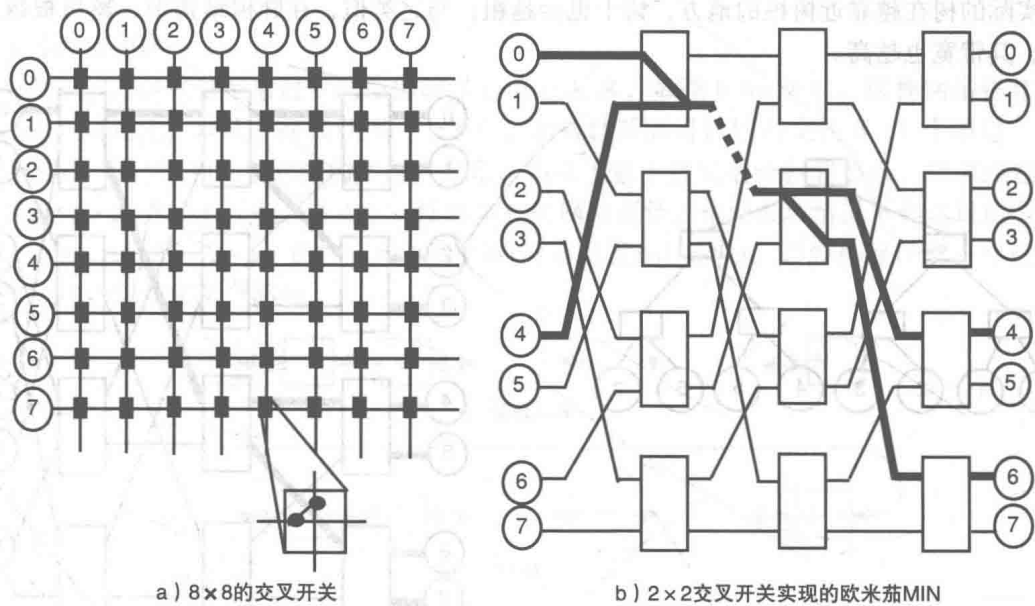


图 6-6 交叉开关和 MIN

在上面例子中的 MIN 中，交叉开关一共包括 3 级。第一级中的交叉开关通过全混洗交换的方式连接到下一级。全混洗交换的连接方式如下：第一级的输出端口从上到下被标记成 0, 1, ..., 7, 这些端口依次连接到下一级的 0, 2, 4, 6, 1, 3, 5, 7 号输入端口，组成了一个全混洗交换操作。全混洗这一命名是根据扑克牌中的洗牌操作得来的：洗牌时把一副牌分为两组，然后交错混洗到一起。基于全混洗的交换网络也被叫作欧米茄网络。

通常情况下，用交换度数为 k 的交叉开关构成的 MIN 需要 $M\log_k N/k$ 个交叉开关。如图 6-6b 中的 MIN，用两个输入端口和输出端口的交叉开关 ($k=2$) 连接了 8 个节点 ($N=8$)，所以需要的交叉开关数量为 $8 \times 3/2 = 12$ 。这个例子也从成本的可扩展性角度解释了为什么 MIN 比单纯的交叉开关结构更具有吸引力，因为此时交叉开关数量的增长和节点数不是平方关系而是对数关系： $O(M\log_k N)$ 。不过，MIN 结构下，数据包从源节点到目的节点所要经过的交叉开关数量更多，需要经过 $\log_k N$ 个，而在 $N \times N$ 的交叉开关结构中，数据包只需经过一个交叉点。

对于节点数较多的网络来说，MIN 的成本要低于交叉开关，不过 MIN 比交叉开关更容易受到竞争的影响。举个例子，输入节点 0 希望和输出节点 4 建立一条路由，输入节点 4 希望和输出节点 6 建立一条路由，这两个路由在图 6-6b 中用粗线进行了标记，如图中所示，虽然这两条路由不会在同一个交叉开关内部竞争，但是它们共享了第一级和第二级交叉开关之间的链路，因此数据包经过这里时会被串行化。

树

不同级之间的交换器连接方式（也即拓扑结构）会影响路由对共享网络资源的占用，并可能造成竞争冲突。现在考虑一类新的称为树的基本间接网络，图 6-7a 展示了一个二叉树互连网络。通常情况下，一个连接 N 个节点、度为 k 的树，其深度为 $\log_k N$ ，即需要 $\log_k N$ 级的交换。树结构的节点在叶子上，从一个节点发送到另一个节点的数据包首先需要传输到树结构中对源节点和目的节点的共同祖先的交换器节点上，然后再从这个交换器节点把数据包下发到目的节点。树结构虽然简单，但是也有缺点，比如它的对分宽度取决于单独的一条链路，假如大部分的网络流量都要经过位于树根部的对分点，那么树的根部将会形成严重瓶颈。缓解这一瓶颈的主流方法是采用胖树（fat tree）结构。在胖树结构中，并不是所有链路都配备一样的带

宽。实际的树在越靠近树根的地方，树干也会越粗；与之类似，在胖树结构中，离树根越近的链路，其带宽也越高。

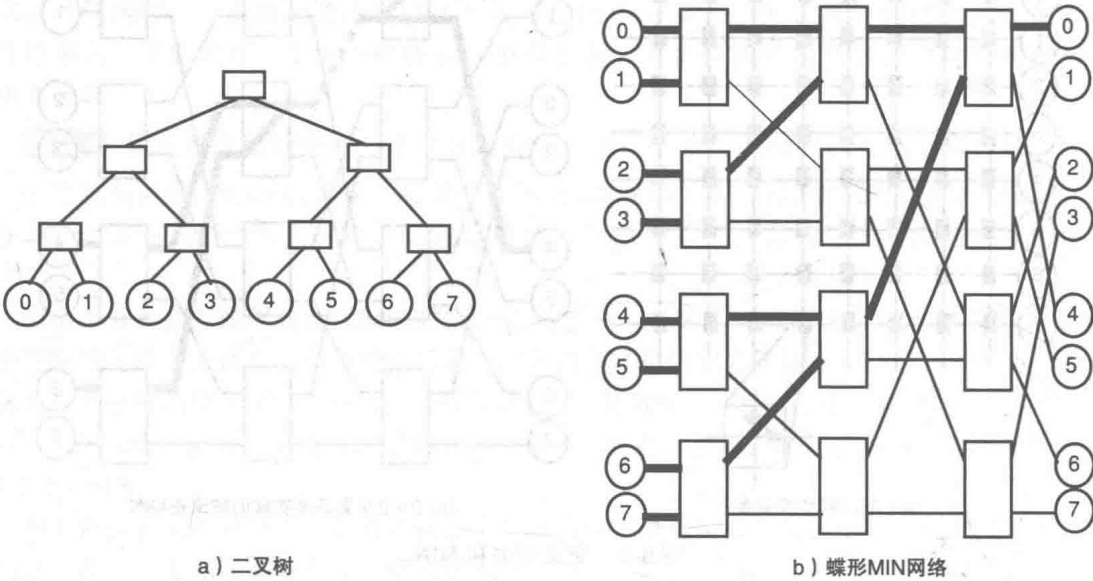


图 6-7 树和蝶形网络

蝶形网络

蝶形网络是一种内嵌树结构的 MIN，在图 6-7b 中加粗显示的是从所有源节点到目的节点 0 的路由，这些路由组成了一个二叉树，从图中容易看到，蝶形网络中树根的数目和目的节点的数目是一样的，虽然树结构中跨越根部的地方可能需要共享一些链路或者交换器。

例 6.5 在蝶形网络找出两个内嵌的逻辑二叉树，要求这两个二叉树不存在任何链路共享，并且路由传输时不会有任何冲突。

这是可能的，比如以目的节点 0 为根节点的树，和以目的节点 7 为根节点的树之间，就没有任何的链路共享。

6.4.2 直接网络

当多个某类设备需要和多个其他类设备连接时，采用间接网络是非常方便的，比如在图 6-1a 中的片上多处理系统，就是使用间接网络将带有一级 cache 的处理器核连接到共享 cache 的多个 bank 上。在实际环境时，任何类型的间接网络都可能被用到。考虑极端情况之一，如果连接的设备只是很少量，那么总线结构就已经足够了。再考虑反向的另一个极端，如果系统中有上百个设备，那么就需要一些特定类型的 MIN 来提供足够的通信带宽和可接受的延迟。

间接网络的一个缺点是设备没有直接彼此相连——请求必须经过一系列的交换器单元，并且交换器的数目通常是固定的，不管设备之间是怎么通信交互的。当设备的数目较多时，使用间接网络通常就不再有优势，而是会倾向于如图 6-2 所示的 mesh 网络一样将设备在网络中分布式摆放，设备紧密地连接上它们对应的交换器单元上。在特殊情况下，当同类设备连接在一起，比如形成计算机主板，或者是片上多处理器系统，这时可以把它当成一个单独的节点，并进一步把它们连接到一个更大的规则互连网络中（比如 mesh 结构），这样有助于利用通信模式的局部性，频繁通信的任务可以分配到相邻的节点，这种类型的互连

网络称为直接互连网络。

线性阵列和环

最简单的直接网络是通过双向链路将节点互连起来，如图 6-8a 所示，这种网络称作线性阵列。和总线相比，总线一次只处理一个消息，而线性阵列可以同时交换 $N-1$ 个消息（假如有 N 个节点）。然而相比于总线的不足之处是，如果消息不是发送给相邻节点，必须经过多个节点，这会导致更长的延迟。事实上，线性阵列的网络直径，也即最坏情况下到达目的节点所需要的跳数，达到了 $N-1$ 。此外，线性阵列的对分带宽也比较差，因为移除任意一个链路都会将线性阵列切分成两个部分。



图 6-8 一维直接网络拓扑结构

为了降低网络直径，提供更高的对分带宽，线性阵列网络可以通过增加一个环绕的链路转换成环网络，如图 6-8b 所示，环网络本质上是将两个端直接连起来的线性阵列网络。增加的这个链路有效地将网络直径缩减为原来的一半，并且将对分带宽扩大为原来的两倍。环绕链路不需要比其他的链路更长，通过合理的布局，可以使得环中的所有链路都一样长度。不过，随着节点数目的增加，环网络中有限的对分带宽依然会限制它的性能。下面介绍其他具有更高对分带宽的直接网络拓扑结构。

Mesh 和 Torus 网络

一个可以大幅增加对分带宽和易于控制延迟的方法是，将节点从单一维度（比如阵列或者环）组织成多维的阵列拓扑结构。如图 6-9a 所示，mesh 结构是线性阵列在二维结构上的简单扩展。假设有 n 行，每行有 n 个节点，在网格中，假如不是外部设备节点，那么这个节点就可以和它周围的 4 个节点互相通信。整个网络的对分带宽为 n ，且和节点数目以平方根的速度增长，而网络直径（即最坏情况下延迟的度量标准）是 $2(N-1)$ ，且也是随着节点数以平方根的速度增加。和一维中形成环的方法类似，可以用水平和垂直的链路将尾端的节点连接起来，从而达到降低网络直径、提高对分带宽的目的。经过这种改进后的 Mesh 结构叫作 torus 结构，如图 6-9b 所示。它将网络直径减半，将对分带宽提升为原来的两倍。Torus 可以认为是在二维的结构中嵌入了一系列水平或者垂直的环。

例 6.6 计算一个连接 64 个节点的 torus 结构的网络直径和对分带宽，假定每个链路的带宽为 b 。

64 个节点被组织成 8×8 的阵列。在 mesh 结构下，最长的路径为对角线上节点之间的路由，而在 torus 结构下，数据包可以从最左下角经过两跳路由到最右上角。与 mesh 结构不同的是，torus 结构的最长路径是，当数据包从一个角落中的节点路由到 torus 中间节点的时候出现的。在一个 $n \times n$ 的 torus 中，这样的数据包一共经过了 $n-2$ 个链路，也就是说，对于 8×8 的 torus 来说需要 6 跳，因此我们可以得到网络直径的值为 6。此外，为了将这个网络分割成两个同构的子网络，必须移除 $2n$ 个链路，因此对分带宽为 $2nb$ ，对于 8×8 的 torus 结构来说，就是 $16b$ 。

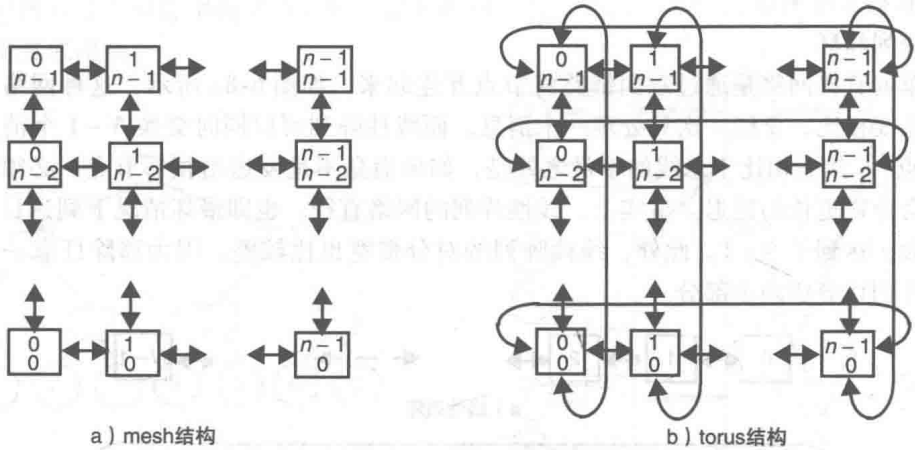


图 6-9 连接 $N = n \times n$ 个节点的二维直接网络拓扑结构

超立方体和 k 元 n 维立方

超立方体 (hypercube) 是一种具有良好延迟和带宽表现的互连网络拓扑结构, 它是一个 n 维的立方体, 且在每个维度只有两个节点。图 6-10a 所示是三维超立方体网络, 我们将这三个维度分别标记成 X, Y, Z 。 X 维是水平坐标轴 (左/右), Y 维是垂直坐标轴 (上/下), Z 维是垂直于 X 和 Y 坐标轴的水平面 (前/后), 如图 6-10a 所示。 N 维的超立方体网络可以连接 $N = 2^n$ 个节点, 网络直径为 $\log_2 N = n$, 对分带宽为 $N/2$ 。相比于 torus 结构, 超立方体网络有更优的网络直径, 其大小随节点数按对数关系增长。此外, 超立方体的对分带宽随节点数线性增长。不过, 每个节点的交换度数 (端口数目) 随着维度线性增长, 这导致当节点数目较多时, 超立方体网络就不再那么有优势了。比如, 当连接 256 个节点时, 超立方体网络必须有 8 个维度, 这样每个节点的交换度数将达到 8。此外, 高维度的超立方体也很难摆放成二维或者三维的形式, 这导致我们很难使用较短的链路。

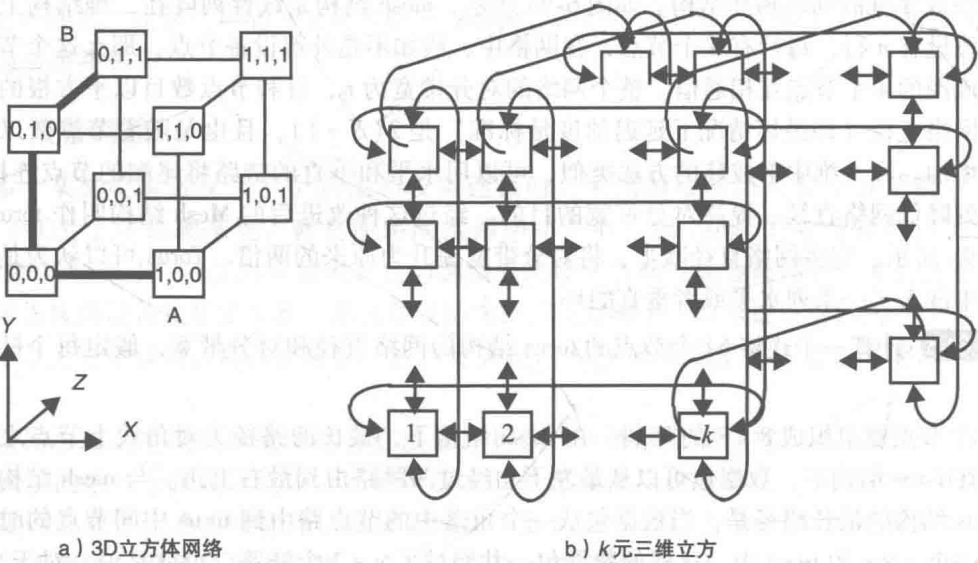


图 6-10 三维直接网络拓扑结构

环、torus 以及超立方体这三种结构都属于直接互连网络拓扑结构中的一类, 叫作 k 元 n 维立方 (k -ary n -cube) 结构, n 代表着维度, k 代表在每一个维度上的节点数。在这种命名方法

中， $n \times n$ 的 torus 可以命名为 n 元二维立方， n 维的超立方体网络可以命名为二元 n 维立方。正如我们将在 6.5 节中介绍的，所有 k 元 n 维立方的拓扑结构都存在一些简单的路由算法。图 6-10b 显示了一个 k 元三维立方结构，它是由 k 个 $k \times k$ 的 torus 结构逐个堆积起来的。在每个平面中，任意两个相邻节点间都有一条链路，在相邻平面的节点之间也有一条链路。和二维 torus 类似，在水平平面和垂直平面的尾端节点之间也存在一条链路。

在本节最后，我们将几种流行的直接网络拓扑结构的拓扑特性进行对比，如表 6-1 所示。有趣的是，我们发现不同的拓扑结构表现出完全不同的特征，比如在网络直径（延迟）、对分宽度（带宽）以及交换度数（成本）等方面都表现得各不相同。这个对比形象地说明了现实中没有放之四海而皆准的拓扑结构，性能、成本和可扩展性之间总是需要互相平衡和取舍的。还有一点需要注意，正如我们之前说过的，诸如网络直径和对分宽度这样的简单拓扑度量标准可以用来对比不同结构下的延迟和带宽，但是在使用时必须十分小心，这些度量标准描述的是最坏情况下的行为，因为它们反映的是距离最远却又频繁通信的节点之间的延迟和带宽。

表 6-1 不同拓扑结构下的 N 节点互连网络特征对比

互连网络	交换度数	网络直径	对分带宽	网络大小
交叉开关	N	1	N	N
蝶形网络 ($k \times k$ 的交叉开关)	k	$\log_k N$	$N/2$	N
k 阶树	$k + 1$	$2 \log_k N$	1	N
线性阵列	2	$N - 1$	1	N
环	2	$N/2$	2	N
$n \times n$ 网格结构	4	$2(n - 1)$	n	$N = n^2$
$n \times n$ torus 结构	4	n	$2n$	$N = n^2$
k 维超立方体	k	k	2^{k-1}	$N = 2^k$
k 元 n 维立方	$2k$	$nk/2$	$2n^{k-1}$	$N = n^k$

6.5 路由技术

这一节我们将介绍把数据包从源节点传输到目的节点的各种路由方法。在 6.5.1 节，我们将回顾 6.4 节中描述的拓扑结构所用到的常见路由算法。路由算法容易出现死锁的问题，不过，通过一些强制的限定条件，死锁的出现是可以避免的（在 6.5.2 节介绍）。6.5.3 节将介绍虚通道技术，这种技术使得路由算法更加灵活，同时还能避免死锁。最后在 6.5.4 节，我们简单介绍一下自适应路由技术。

6.5.1 路由算法

不管采用什么样的交换策略（电路交换策略或者是数据包交换策略），路由消耗的时间都是在数据包传输的关键路径上，这会对延迟产生直接影响，因此，路由算法一定要尽可能简单，这样才能尽可能减少开销。在 LAN 和 WAN 网络中，一种简单的路由算法是在每个路由器中执行查表操作。由于开销太大，这种基于查表的方法在 OCN 以及 SAN 网络中很少采用。在这一节，我们介绍一些简单并且对延迟影响较小的路由算法。

交换器是所有互连网络的基本组成部件，因此在交叉开关网络中，应该首先明确数据包是如何从输入端口路由到输出端口的。假如交换度数为 k ，那么确定数据包从输入端口到输出端口的路由需要 $\log_2 k$ bit 来表示，对于 2×2 的交叉开关来说，也就是说需要 1bit 来表示路由。例如，对于图 6-6b 所示欧米茄网络中的 2×2 交叉开关网络，一个地址位就能决定数据包是送往

上面的端口还是下面的端口。

在 MIN 中,如图 6-6b 中的欧米茄网络,可以使用目的地址作为路由地址将数据包从节点 A 路由到节点 B。带有 $N=2^n$ 个节点的欧米茄网络,数据包的目的地址为 $D = \langle d_{n-1}, d_{n-2}, \dots, d_0 \rangle$, 路由算法如下:从源节点到目的节点被标记为 0 到 $n-1$ 共 n 级,在第 $n-1-i$ 级中,目的地址 (d_i) 的第 i 个比特位用来决定数据包是被路由到上面的输出端口 (路由比特位为 0) 还是下面的输出端口 (路由比特位为 1)。因此,当数据包从源节点路由到目的节点时,目的地址对应的比特位依次从最高有效位查找到最低有效位。

例 6.7 在图 6-6b 中的欧米茄网络中,通过路由地址确定从节点 4 到节点 6 的数据包路由过程。

图中目的节点 6 对应的地址是 110, 由于两个最高有效位都为 1, 因此数据包在前两级均被路由到下面的输出端口, 最低有效位为 0, 因此在最后一级, 数据包路由到上面的输出端口。图 6-6b 显示了数据包从节点 4 传输到节点 6 的路由过程。

针对图 6-7b 所示的蝶形网络进行数据包路由时, 我们考虑另外一种算法: 路由地址使用将源地址和目的地址按位异或得到的相对地址, 而不是直接使用目的地址。假如源和目的地址分别为 A 和 B , 那么路由地址 $R = A \text{ XOR } B$, XOR 表示按位异或。在包含 $N=2^n$ 个节点的蝶形网络中, 路由算法如下: 假定从源到目的节点之间的 n 级分别标记为 $0 \sim n-1$, 源和目的地址异或得到的路由地址 $R = \langle r_{n-1}, r_{n-2}, \dots, r_0 \rangle$, 这个路由地址 (r_i) 的第 i 个比特位用来控制第 i 级的路由, 这位为 0 时, 数据包就采用直通过路的方式; 为 1 时, 就采用交错路由的方式。假如数据包从 0 号输入端口进入并且路由器设置成直通方式, 那么数据包将发送到 0 号输出端口。如果交换器设置成交错方式, 那么 0 号端口 (上面的输入端口) 进入的数据包将从 1 号端口 (下面的输出端口) 输出, 而 1 号端口 (下面的输入端口) 输入的数据包将从 0 号端口 (上面的输出端口) 输出。

事实上, 基于源和目的相对地址的路由算法在包括线性阵列、mesh 以及超立方体在内的大量互连网络中被广泛采用。 $n \times n$ 的 mesh 网络中有一种非常流行的路由算法叫作维度优先路由, 这就是一个很好的例子。在这种路由算法中, 所有节点使用 x 和 y 坐标标记。比如, 在图 6-9a mesh 中的左下角的节点坐标为 $(0, 0)$, 右上角的节点坐标为 $(n-1, n-1)$ 。从节点 $A(x_A, y_A)$ 传输到节点 $B(x_B, y_B)$ 的数据包的路由地址为相对地址 $R = (x_B - x_A, y_B - y_A)$ 。在维度优先路由算法中, 数据包先在 x 维上进行路由, 然后在 y 维上进行路由, 这种方法可以扩展到任意维度。

每个交换单元上的路由决策非常简单, 只要相对地址的 x 维坐标不为零, 那么数据包就在 x 维上路由, 如果 x 坐标比零大, 那么数据包在 x 的正向路由, 并且相对地址的 x 坐标递减。如果 x 坐标比零小, 那么数据包在 x 的负方向路由, 并且相对地址的 x 坐标递增。当 x 坐标变成零, 那么数据用同样的方式在 y 维上路由。当相对地址中的 x 和 y 坐标都变成零了, 数据包就到达了目的地并被注入目的节点。可以看到, 交换器中的路由只需查看相对的路由地址并进行递增或者递减, 这种路由开销很小。

例 6.8 请描述在图 6-2 的 mesh 结构中, 数据包从节点 A 传输到节点 B 的路由过程。

节点 A 和 B 的坐标分别为 $(1, 0)$ 和 $(3, 1)$, 相对地址为 $(2, 1)$, 因此, 数据包先在 X 维正向 (东) 跳两步, 然后左拐向 Y 维正向 (北) 传输达到节点 B。

对于超立方网络, 在每个维度上只有两个节点, 和蝶形网络一样, 相对地址由源地址和目的地址按位异或得到。以图 6-10a 中所示的三维超立方网络为例, 数据包从节点 $A(1, 0, 0)$ 传输到节点 $B(0, 1, 1)$, 相对地址 $R = (1, 0, 0) \text{ XOR } (0, 1, 1) = (1, 1, 1)$ 。因此, 数据包

先在 x 维移动，然后在 y 维移动，最后在 z 维上沿着加粗路线移动。这种路由算法是专门应用于超立方网络的一种特殊维度优先路由算法，称为 e -立方路由算法。

6.5.2 死锁避免和确定性路由

除了实现简单，维度优先路由还可以避免死锁。死锁指的是多个请求之间互相阻塞导致谁也无法继续执行。图 6-11a 显示了在 msh 网络中出现死锁的一种情况，图中每个节点都试图向它的对角节点发送一个数据包。发送数据包需要占用一定资源，包括一条物理链路和下一个交换器用于存储流控单元的缓冲空间，比如从节点 $(0, 0)$ 向节点 $(1, 1)$ 发送数据包时，首先需要在节点 $(0, 0)$ 和节点 $(0, 1)$ 之间分配一条链路，并且在节点 $(0, 1)$ 的交换器中分配一块缓冲空间。节点 $(0, 1)$ 上会出现两个数据包同时竞争同一个输出端口的情况，这时只有一个能够竞争胜出。在这里是节点 $(0, 1)$ 注入的数据包胜出了，因此来自节点 $(0, 0)$ 的数据包会被阻塞。其他节点也会重复类似的行为，比如，节点 $(0, 1)$ 注入的数据包也会和节点 $(1, 1)$ 注入的数据包竞争节点 $(1, 1)$ 的同一个输出端口，前者竞争失败被阻塞，最终所有节点形成一个循环依赖链，使得所有的数据包都被阻塞住。

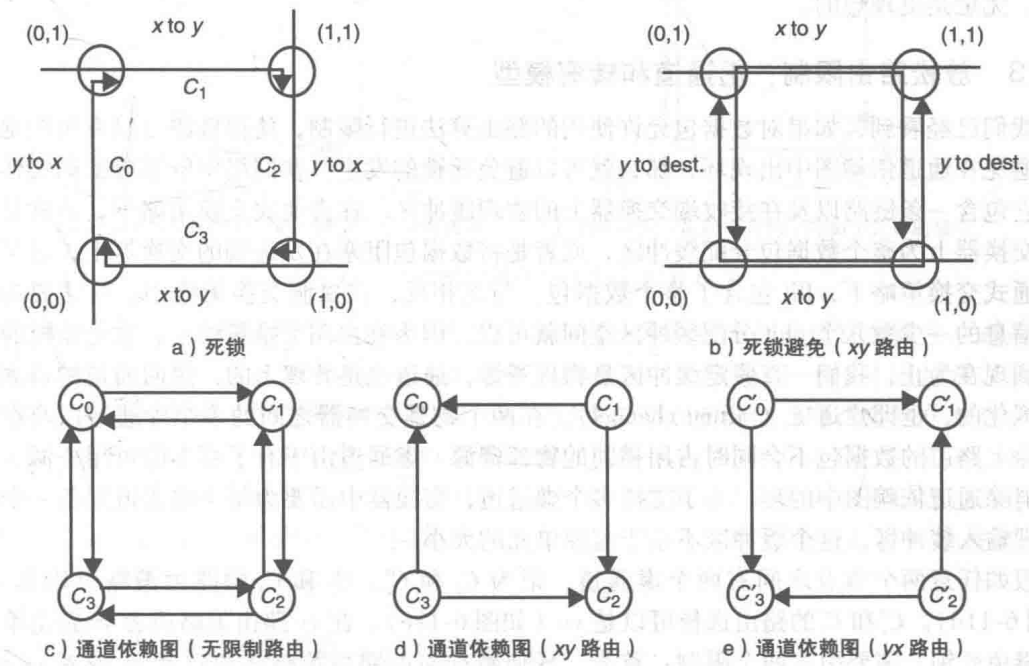


图 6-11 死锁和通道依赖图

上面的情况之所以出现死锁，其问题在于，当节点 $(0, 1)$ 注入的数据包按照先 x 维后 y 维的策略路由时，节点 $(0, 0)$ 注入的数据包却是先 y 维后 x 维路由。如果所有的数据包都是先 x 维后 y 维路由，并且链路是双向的，那么死锁就不会发生，如图 6-11b 所示。和图 6-11a 不同的是，节点 $(0, 0)$ 注入的数据包现在通过节点 $(1, 0)$ 路由到目的节点 $(1, 1)$ ，达到目的节点后就可以从 IN 中删除了。因为数据包有规律地先从 x 维路由，再从 y 维路由，这一过程不会建立循环依赖链，因此死锁也就不会发生。

对上述问题的定义可以这样描述：IN 是由通道组成，而通道是两个相邻交换器之间的互连通路，以及接收端交换器中用来保存 flit 的缓冲空间。通道是当数据包从一个交换器传输到另一个交换器时分配的资源，它和链路有明显区别：通道是一种逻辑连接，而链路是一种物理连接，在同一个物理链路上可能会有多个通道。实际上，这也是图 6-11b 中所默认的假定，通

过对链路的分时复用,两个数据包可以在相同物理链路的两个不同方向上同时传输。

分析是否会出现死锁方法之一是借助通道依赖图,图中的顶点代表通道,两个通道 A 和 B 之间的有向边表示占据通道 A 的数据包可以随后路由到通道 B 。图 6-11a 中通道用 C_0 、 C_1 、 C_2 、 C_3 表示, C_i 和 C_j 之间的边表示根据路由数据包先使用 C_i 然后使用 C_j 。在图 6-11b 中,数据包可以首先在节点 $(0, 0)$ 和节点 $(1, 0)$ 直接路由,然后再在节点 $(1, 0)$ 和节点 $(1, 1)$ 之间路由。这一合法路由在图 6-11d 中表示成通道依赖图中 C_3 和 C_2 之间的一条边。如果是无限制路由,那么会形成图 6-11c 所示的通道依赖图。如果这个图是有环的,那么就意味着可能发生死锁。与之相反,如果通道依赖图是无环的,那么死锁就不会发生。正如维度优先路由策略那样,如果对路由进行限制,使得数据包必须先在 x 维路由然后再在 y 维路由,那么通道依赖图就会是无环的。图 6-11d 和图 6-11e 分别显示了 xy 路由和 yx 路由算法下的通道依赖图,它们都不存在环。

虽然限制路由选择可以避免死锁,但它会导致互连网络的带宽无法充分利用。比如,在图 6-11a 中,节点 $(0, 0)$ 注入的数据包要发送到节点 $(1, 1)$,在 xy 的路由策略下,它必须首先经过 C_3 通道,假如 C_3 这时候处于忙碌状态,这时候如果能使用通过 C_0 和 C_1 的其他路由算法,无疑是更理想的。

6.5.3 放松路由限制:虚通道和转弯模型

我们已经看到,如果对数据包允许使用的路由算法进行限制,使得资源可以有组织地分配从而避免在通道依赖图中出现环,那么就可以避免死锁的发生。在网络中资源分配的实体是通道,它包含一条链路以及在接收端交换器上的物理缓冲区。在直通式交换策略下,通常是在接收端交换器上为整个数据包分配缓冲区,或者是将数据包阻塞在发送端的交换器上,这是因为在直通式交换策略下,flit 包含了整个数据包。与之相反,在虫洞交换策略中,只需要为携带路由信息的一少数几个 phit 分配缓冲区空间就可以,因为在虫洞交换策略下,这是流控的。

到现在为止,我们一直假定缓冲区是物理资源,通道也是物理上的,然而通道的概念是可以虚拟化的,也即虚通道(virtual channel)。在两个物理交换器之间的多个虚通道以及在不同虚通道上路由的数据包不会同时占用相同的物理资源。虚通道由于有了更多的可用资源,因此可以消除通道依赖图中的环。为了支持多个虚通道,交换器中需要为每个虚通道配备一个专门的物理输入缓冲区,这个缓冲区不小于流控单元的大小。

假如任意两个节点之间有两个虚通道,记为 C_i 和 C'_i 。 C_i 和 C_j 的路由策略可以是 xy 的(如图 6-11d)。 C'_i 和 C'_j 的路由选择可以是 yx (如图 6-11e)。在 xy 路由策略或者 yx 路由策略下要想避免死锁,需要引入两个限制:首先,任何数据包的路由策略必须符合 xy 或者 yx 路由;其次,只能允许数据包从 xy 到 yx (或者 yx 到 xy) 切换一次。比如,假定所有的数据包一开始都使用 xy 路由策略,它们中的任何一个都可以根据需要自由地切换成 yx 路由,但提前是它们之后不会再切换回 xy 路由。这可以看成是维度优先路由策略从二维到三维的一个扩展。比如,数据包在第一组虚通道中首先用 xy 路由,然后数据包可以在第二组分配给 yx 路由的虚通道中切换到 yx 路由。可以看到,基于 C 和 C' 通道构造的通道依赖图中是没有环的,因此不会出现死锁。可见虚通道可以在避免死锁的前提下消除一些路由限制,因此可以在流量热点区域或者存在故障的交换区域提供更加灵活的数据包路由策略。

虚通道是一种通过移除通道依赖图中的环来避免死锁的通用设计方法,此外,也还有其他一些方法可以消除死锁,一种常用的方法是转弯模型。还是以二维 mesh 网络为例,完全无限制路由算法允许数据包采用 8 个不同的转弯方向;而 xy (或者 yx) 路由策略将转弯方向的数目限制到 4 个:从 $+/-x$ 到 $+/-y$ (或者从 $+/-y$ 到 $+/-x$),如图 6-12a(或者图 6-12b)所示。

这里的 + 和 - 符号代表数据包的传输方向，比如 $-x$ 表示朝 x 的负方向。那么接下来的问题是，在避免死锁的前提下，能否在转弯数目上做尽可能少的限制。

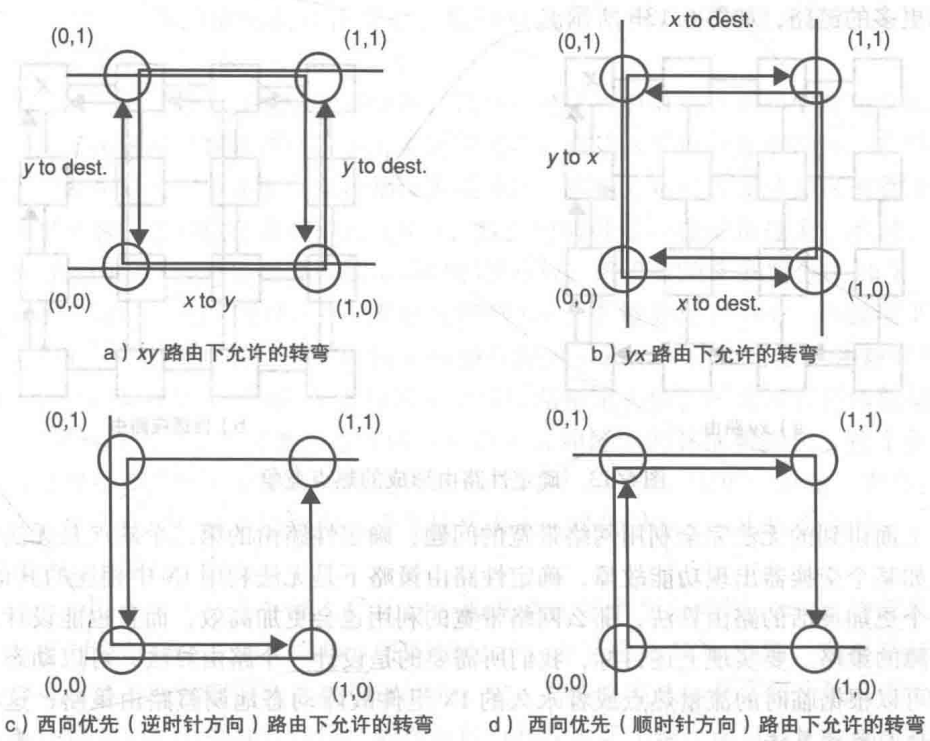


图 6-12 不同路由算法中允许的转弯情况

基于转弯模型，可以实现很多相比维度优先路由算法限制更少的其他路由算法。西向优先路由算法就是一个例子，它支持的合法（无死锁）转弯方向如图 6-12c 和 6-12d 所示。从算法命名中可以看出，在西向优先路由中，假如目的节点在源节点的西侧，那么数据包首先在向西的方向上路由。比如在图 6-12c 中，可以看出，假如数据包从节点 $(1, 0)$ 到节点 $(0, 1)$ ，路由是不允许经过节点 $(1, 1)$ 的。正如图 6-12d 所示，数据包必须先向西移动通过节点 $(0, 0)$ ，然后再向北移动。在西向优先的路由选择中，8 个可能的转弯方向中只有两个被禁止了，这比 xy 或者 yx 的路由算法更加灵活。

6.5.4 进一步放松的路由算法：自适应路由

从前面的介绍中可以看到，诸如维度优先路由这样的确定性路由算法的一个好处是，它能保证不出现死锁；另一个好处是，它能保证所选的路径是从源节点到目的地节点的所有路径中最短的。实际上，确定路由算法总是可以保证路由在一定数量的步数之后能够完成。如果要支持更多的合法路由，一个基本的问题是要保证数据包朝着目的节点的方向前进，如果做不到这一点，那么就可能出现活锁 (livelock)，在最坏的情况下，活锁可能会导致数据包一直在 IN 中传输但却永远到达不了目的节点。

确定性路由算法也有缺点。第一个缺点是，当遇到网络竞争的时候，无法利用网络中的空闲资源来重新路由数据包。事实上，它们的确定性路由特性可能会促使网络中形成流量热点区域。我们用图 6-13 来解释这一现象，图中显示了所有的节点都向节点 X 发送一个数据包的情况。假如 X 包含了一个全局变量，而在临界区内的所有节点都要对这个变量进行递增，上述情况就会出现。图 6-13a 的箭头显示了在 xy 路由选择策略中，数据包从网格的底部那一行开始路

由。可以看到这样一个明显的现象，那就是维度优先路由中的确定性路由限制很容易把数据包都集中在可用链路的一个很小子集中。与之相反，如果数据包的路由选择没有限制，那么流量可以使用更多的链路，如图 6-13b 所示。

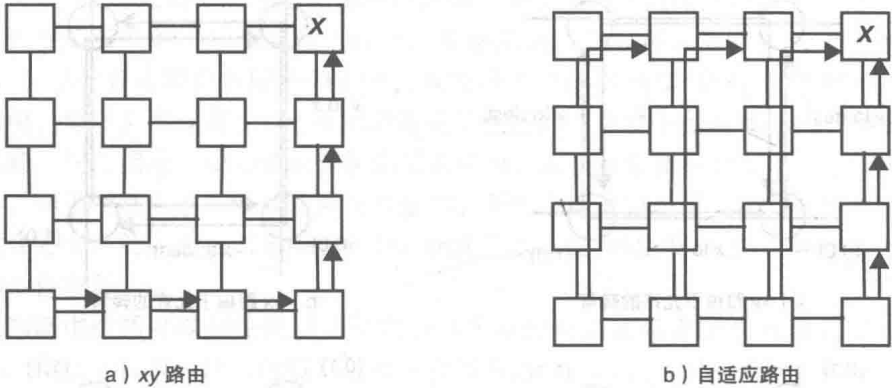


图 6-13 确定性路由形成的热点竞争

除了上面讲到的无法完全利用网络带宽的问题，确定性路由的第二个缺点是无法应对网络故障。假如某个交换器出现功能故障，确定性路由策略下是无法利用 IN 中相连的其他路由的。如果有一个更加灵活的路由算法，那么网络带宽的利用也会更加高效，而且也能设计出有效应对网络故障的策略。要实现上述目标，我们所需要的是设计一个路由算法，可以动态更改路由策略，它可以根据临时的流量热点或者永久的 IN 组件故障动态地调整路由策略，这种路由算法称作自适应路由算法。

在本节中，我们已经描述了利用虚通道构建自适应路由算法的一个大体框架。借助于虚通道，我们有可能设计出一种无死锁且能灵活应对流量热点和 IN 组件故障的路由算法。比如当数据包到达的交换器由于临时流量高峰而出现激烈竞争时，或者原先路由上的交换器出现故障时，利用两组虚通道就可以从 xy 路由切换到 yx 路由（但是不能再切换回来）。

通常情况下，自适应路由算法更加复杂，这可能会对网络延迟产生负面影响。

6.6 交换架构

本章最后讨论的是交换单元（即交换器）的设计，这部分内容非常重要，因为交换器实现了从源节点到目的节点路由数据包的所有功能。

图 6-14 显示了一个 4×4 交换器的主要部件。交换器的中心部件是交叉开关（在 6.4.1 节介绍过），交叉开关的每个输入和输出端口都连接着一个缓冲区。交换器使用在 6.3 节和 6.5 节介绍过的任意一种交换策略和路由技术，将数据包从输入端口路由到输出端口。虽然交换器的功能看起来比较简单，但是要满足低延迟和高带宽的要求，还是需要仔细斟酌所有的设计选项。为了描述得更清晰，我们对一个通过交换器的数据包进行跟踪。在直通式交换策略中，只有当输入端口有足够的缓冲区来缓存整个数据包的时候，数据包才从一个交换器传输到另一个交换器。

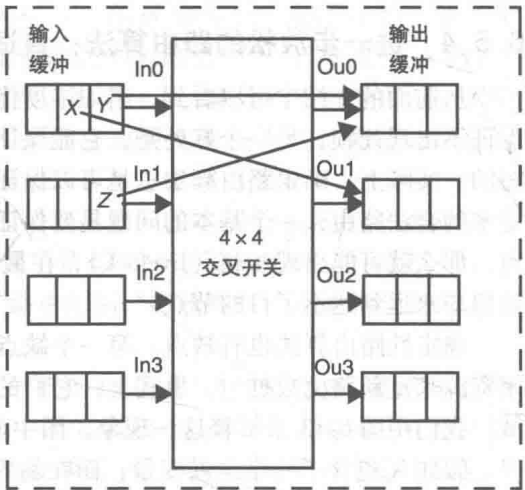


图 6-14 4×4 交换器架构

与之相反,在虫洞交换策略中,只要缓冲区能够存下带有路由信息的少数几个 phit 时,数据包才可以传输到下一个交换器。当路由信息到达交换器后,数据包就开始仲裁交叉开关的输出端口。当交叉开关的输出端口就绪时,数据包就路由到选中的输出端口,并决定是否需要先缓存下来。

在每个交换器设置缓冲区是非常重要的,这种设计可以有效缓解现实中难免会出现的流量高峰问题。在交换器内设置缓冲区有 3 个不同的选项:在输入端口设置缓冲区,在输出端口设置缓冲区,同时在输入端口和输出端口都设置缓冲区。在输入端口设置缓冲区的优点是,只要接收交换器有足够的空间缓存流控单元 (flit),数据包就可以一直发送过去。不过,数据包要想继续往前传输的话,还需要再赢得输出端口的仲裁权。假设有两个数据包 X 和 Y ,现在都缓存在图 6-14 所示的 $In0$ 输入缓冲区中, X 被分配到 $Ou1$, Y 被分配到 $Ou0$ 。再假设 Y 不能进行路由,因为和 $In1$ 对应缓冲区中的数据包包 Z 也被分配到 $Ou0$,并且 Z 已经在仲裁时被选中了。这个时候, Y 就会阻塞住 X ,即使 X 本身其实是可以路由出去的,因为没有任何数据包分配给 $Ou1$ 。除了阻塞住 X 外, Y 还可能阻塞住同一个输入缓冲区中的其他数据包。这个例子也暴露了输入端口设置缓冲区的一个缺点,称之为线头 (head-of-line, HOL) 阻塞。需要注意的是,只要不存在竞争,数据包就不用缓存,缓存只是用来应对竞争的——出现竞争时,交换器可以暂时缓存对应交换策略下的 flit。

这个例子也暴露了另一个性能上的问题,那就是在每个交换周期, $n \times n$ 的交换器内可能会有 n 个数据包竞争同一个输出端口。理想情况下,为了避免 HOL 阻塞,交换器和输出缓冲区执行的速度应该是交换器接收数据包速度的 n 倍。解决方案之一是在每个输入端口上设置和输出端口数一样多的输入缓冲,这样,数据包可以在注入某个输出端口对应的输入缓冲之前就建立好路由。但是这个解决方法的问题是其可扩展性不好,对于一个 n 个输入端口和 n 个输出端口的交换器来说,需要 n^2 个输入缓冲区。

6.5.3 节中介绍了一些更灵活的路由算法,可以通过支持虚通道提高网络的利用率。在这种情况下,每个虚通道都必须配备一个专门的物理输入缓冲区。比如,假定图 6-14 的交换器架构支持两个虚通道,那么交叉开关的每一个输入端口都需要两个输入缓存区。

为了获得更高的数据包吞吐率和带宽,有必要充分利用数据包处理过程中的大量并行性。每个 phit 可以分多步处理:首先,对携带路由信息的数据包头部进行解析,并确定这个数据包路由的输出端口;然后,执行获取对应输出端口的仲裁;再后,头部被转移到输出缓冲区;最后,当目的节点交换器有足够空间缓存该头部信息对应的 flit 时,数据包头部被传输到目的交换器中。数据包的其他 phit 依据设定的交换策略紧跟头部 phit 传输。因此,交换器可以通过流水的方式有效支持这些传输步骤。为了保证较高的流水线时钟频率,需要确保每个传输步骤尽可能简单,这也是为什么使用低延迟的简单路由策略会如此重要。除了每个数据包的处理过程可以并行流水外,多个数据包之间也可以并行处理,因为不同数据包之间没有任何依赖。

只要数据包之间没有竞争,数据包内的流水以及数据包之间的并发这两种并行机制通常都能带来更高的处理带宽。拥塞控制可以解决数据包之间的冲突。在图 6-15 中,增加了交换器之间的握手信号,用来支持链路级流控。接收端交换器提供了一个 Rdy 的握手信号,用来告知发送端交换器它有足够的缓冲区用于数据接收。只要 Rdy 有效,发送端交换器就可以传输数据包中的 phit。在虚拟直通式交换策略中,有效的 Rdy 信号表示接收端的缓冲区至少可以容纳整个数据包;在虫洞交换策略中,则表示接收缓冲区可以足够容纳路由所在 phit。而如果出现竞争,某些交换器中的 Rdy 信号会被无效,此时对应的流量可能需要原路

返回到源节点。

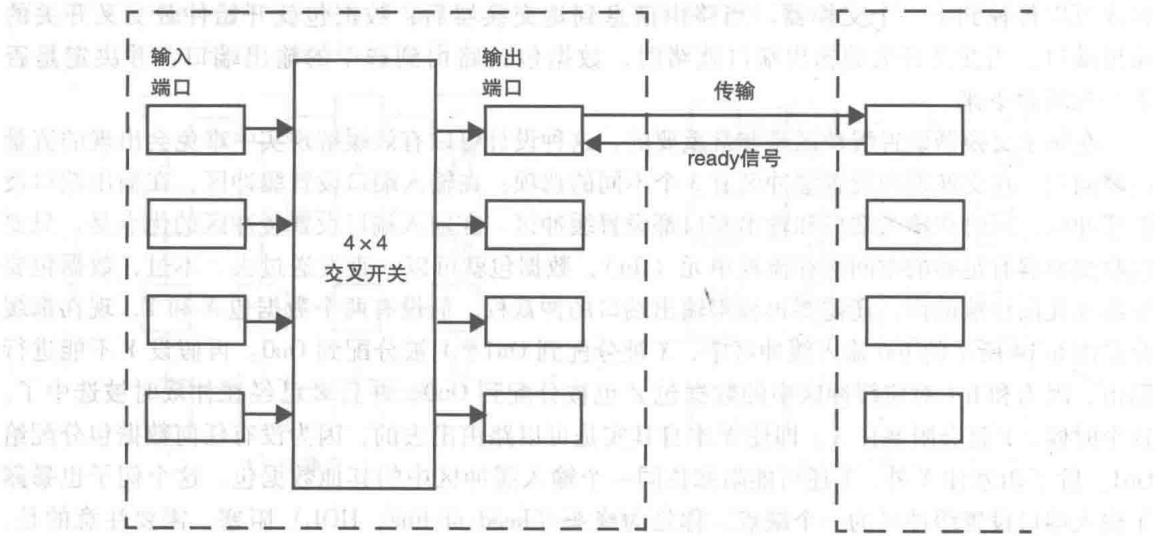


图 6-15 相邻两个交换器的输出端口和输入端口之间带流控的 4 × 4 交换器架构

习题

- 6.1 在图 6-2 所示的 mesh 互连网络中，假定网络时钟频率是 1GHz，phit 的大小为 8bit，路由地址占用 2 个 phit，数据包的有效负载为 128bit，信封是 16bit。
- (a) 假设发送端和接收端的开销为零，使用的是直通式交换策略，从左下角的节点发送一个数据包到右上角节点的端对端网络延迟是多少？
 - (b) 有效带宽是多少？
 - (c) 假设发送端和接收端的开销分别为 64ns 和 128ns，这种情况下的有效带宽又是多少？
- 6.2 使用和习题 6.1 中相同的网络参数、拓扑结构、数据包大小设定，使用存储转发交换策略。
- (a) 假设发送端和接收端的开销为零，计算从左下角的节点发送一个数据包到右上角节点的端对端网络延迟。
 - (b) 使用存储转发交换策略时，在每个交换器中的缓冲区最小是多大？
 - (c) 有效带宽是多少？
- 6.3 在一个 16 × 16 的 torus 互连网络中，计算下列互连网络特性：
- (a) 网络直径。
 - (b) 对分带宽，假设每个链路的带宽为 100Mbits/s。
 - (c) 每个节点的带宽。
- 6.4 一个设计团队正在评估应该使用非流水的（原子形式）还是流水的（分离事务形式）的总线来连接私有 cache 子系统和二级共享 cache，分析主要基于如下数据：
- 处理器单发射，时钟频率为 2GHz。
 - CPI（每条指令消耗的时钟周期）为 1，不考虑访存引起的阻塞。
 - 私有 cache 中每个指令的失效率为 1%。
 - 二级 cache 的访问时间为 20ns。
 - 在一个总线周期内，总线可以发送一个地址并返回整个 cache 块。
 - 总线周期时间为 2ns。
 - 处理器核数是 4。
- (a) 假如使用非流水的总线，总线的利用率是多少？

- (b) 假如使用流水的总线，总线的利用率是多少？
 - (c) 基于上述数据，你的建议是什么？
- 6.5 假定需要设计一个多级互连网络用于连接两类不同的节点，每一类分别有 64 个节点，可用的交叉开关大小从 2×2 到 64×64 不等，不同交换度数下的交换延迟如表 6-2 所示。
- (a) 使得端对端飞行时间达到最小的交换度数是多少？
 - (b) 根据问题 (a) 中确定的交换度数，使用混洗交换互连模式构建一个 MIN。
 - (c) 标记出使用了同一条链路或者使用了相同交换资源的两条路由。

表 6-2 不同交换度数下的交换延迟

交换度数	交换周期时间 (ns)
2	12
4	15
8	30
64	70

- 6.6 设计团队正在评估如何使用 $n \times n$ 的 torus 网络或者 n 维超立方网络来分别搭建 4 节点、16 节点、64 节点和 256 个节点的互连网络。在本习题中，通过对比这两种拓扑结构的网络直径和对分带宽，可以更好地理解应该如何在它们之间进行折中。
- (a) 在多大的网络规模情况下，超立方网络会比 torus 网络有更高的对分带宽？
 - (b) 在问题 (a) 得到的网络规模下，计算对应的网络直径和交换度数。
 - (c) 谈一谈你对这两种拓扑结构优缺点的理解。

假定有一个四元三维立方互连网络，为这个网络设计一个维度优先路由算法。说明路由由地址是如何生成的，数据包从源节点 (x_1, y_1, z_1) 到目的节点 (x_2, y_2, z_2) 是如何路由的。

一致性、同步与存储一致性

7.1 概述

本章主要探讨共享内存多处理器系统中值的正确性和可靠通信问题，这类系统中的存储相关正确性特性包括：cache 一致性，存储一致性模型（后续也简称为存储模型），以及同步原语的可靠执行。由于 CMP 也是共享内存的多核系统，所以本章的内容不仅包括第 5 章讲到的 SMP 系统以及大型 cc-NUMA 和 COMA 系统的正确性问题，也包括第 8 章中将会介绍的 CMP 系统。

共享内存多线程程序的正确性必须独立于每个线程的相对执行速度，因为很多不可预测的事件都可能中断线程的执行，比如 DVFS（动态电压和频率调整）、硬件和软件资源冲突、中断、异常、内核活动、线程调度、数据分配延迟，以及与其他运行程序的交互等。但是假如多线程程序是服务于某些特定的、时间高度可预测的机器，并且程序出于正确性要求已经考虑了时间因素（比如，在实时系统中），那么本章的很多结论都不再适用。换句话说，本章针对的目标软件是为通用或多用途机器编写的可移植共享内存多线程程序，也包括操作系统内核程序。

从硬件的角度来看，存储模型是本章所讨论的共享内存系统中最重要的一部分。存储模型是 ISA（指令集架构）规范的一部分。简而言之，存储模型决定了共享内存多处理器中访存指令（load 和 store）交错的合法性，也就是说，它决定了一个线程的访存操作何时能被其他线程看到。这对整个系统的正确性至关重要，也将直接影响结构、硬件和软件的设计。存储模型提供了硬件和软件设计人员之间的简单接口。软件设计人员（编译器开发人员，汇编代码程序员和操作系统程序员）必须按照存储模型的基本规则来编写代码，以保证代码在任何支持该存储模型的机器上都能够正确执行。同样，只要架构设计人员和硬件设计人员设计的硬件符合存储模型（不管实现起来多么复杂），就可以不用考虑对应机器上运行软件的复杂性和多样性。

线程间的同步问题是从存储模型中独立出来的一个问题，因为共享内存所支持的通信机制比较简单且有限，因此有时候就需要线程间的同步。自从分时操作系统出现之后，同步变得更加必要，分时操作系统通过定期的进程上下文切换以实现在单处理器上执行多个并发进程，多个进程通过分时机制复用计算机的所有资源。多处理器系统中的线程间同步比分时单处理器系统中的线程同步更加重要，因为前者需要维护同时执行的多个线程内的指令顺序，而不是一次只有一个在执行。虽然存储模型和同步是两回事，但它们也存在关联性。程序中的显式同步规定了线程间的指令的生效顺序，因此，一些存储模型的定义也利用了显式同步点的这一特性。此外，因为同步指令也是访存指令，所以存储模型也必须为不同线程之间带 load 和 store 的交错同步指令明确顺序。

本章的目的不是探讨复杂的形式化问题或者具体的硬件实现（具体实现在第 5 章中有介绍），同样，也不会详细介绍所有的存储模型，或揭示有哪些硬件层面的“技巧”可以用来确保硬件和存储模型的匹配，这类“技巧”往往是高度依赖于所实现的模型和特定的体系结构的。与之相反，本章的目标是让读者对多处理器中存储访问的含义和对实际硬件的影响有更直观的感受，以便更好地指导初始设计以及后续的设计修订。这种直觉越正确，就越容易做出更

好的设计，后续所需的迭代次数和验证工作也就越少。

本章的重点是架构和硬件，高层次的软件问题（比如编译器和编程语言的影响）会根据需要简单介绍。我们将使用一些抽象的硬件组件模型，来解释一致性（coherence）、store 原子性、顺序一致性以及其他的存储模型。我们将使用大量例子来讲解，并在介绍例子的过程中定义一些重要概念。通过对这些重要例子的学习，读者应该可以推广到其他的实际情况中去。

本章包含的内容如下：

- 背景。介绍共享内存通信模型和共享内存架构中的硬件组件。为什么在现代的共享内存多处理器系统中内存的正确性属性很难保证？这部分内容将在 7.2 节中介绍。
- 一致性的不同层次。7.3 节主要介绍纯一致性和 store 原子性之间的区别。
- 存储模型和顺序一致性的介绍。包括最基本的存储模型，以及如何通过 store 同步来确保顺序一致性。这部分内容将在 7.4 节介绍。
- 7.5 节主要介绍线程同步和 ISA 级同步原语。
- 7.6 节主要介绍基于硬件效率的放松存储模型和依赖于同步的放松存储模型，以及支持原子 store 共享内存并且装备 store buffer 的静态顺序流水线的简单多处理器结构中的存储模型实现。
- 7.7 节主要介绍支持原子 store 共享内存并且具有推测执行、乱序处理器核的多处理器存储模型的实现，以及存储模型的推测执行冲突。

7.2 背景

为了更好地理解本章的内容，这一节将集中介绍所有的背景知识，部分内容可能会和前面的章节有所重复，首先我们从共享内存的通信模型开始介绍。

7.2.1 共享内存通信模型

在共享内存的多处理器中，不同线程通过对共享内存进行常规 load 和 store 操作来完成值的隐式通信。“隐式”意味着不像消息传递系统那样会在代码中显式包含信号通信指令，所以，单纯通过查看代码无法识别通信发生的地方。图 7-1 表明了从生产者线程 T1 到消费者线程 T2 的一个通信过程，T1 在共享内存的 A 地址存储了一个值，T2 从地址 A 读取这个值。这和消息传递系统差别很大，后者会在代码中插入显式的发送和接收原语，以此将数据从一个线程传输到另一个线程，这里的发送或者接收消息都可能会引起线程同步。

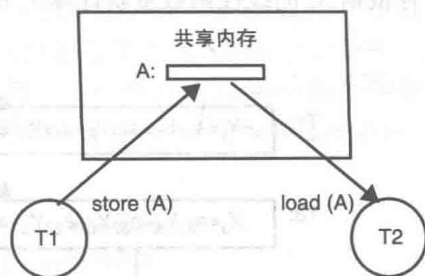


图 7-1 共享内存通信

在图 7-1 中的生产者/消费者示例中，如果 load 需要返回的是 store 的值，那么就必须添加同步操作。事实上，假如没有同步操作，那么 load 操作可能在 store 操作之前或者之后执行，因为每个线程的执行速度是难以预测的。程序的不同执行可能导致 load 操作返回不同的值。

在绝大多数情况下，我们希望多线程程序的表现具有确定性，并且在每次执行时都能得到相同的值。我们也期望，给定一个同样的输入集，多线程程序和它的单线程程序版本应该得到一样的结果。为了强制 store 操作在 load 之前发生，线程 T1 必须通知 T2，由 store 操作产生的 A 的新值已经就绪，而 T2 必须等待这个通知信号才能读取 A 的值。

反过来同步策略的成功取决于对存储模型的深入理解。比如，程序员可能要通过一个二进制标记（flag）来同步 store 和 load 操作，如下面的代码所示：

```
INIT: A=FLAG=0
T1      T2
A=1
FLAG=1
while(FLAG==0);
Print A
```

在这个程序中，程序员期望 A 的值打印出来总是 1。在共享内存系统中，除了将一个线程产生的值可靠传播到另一个线程，还需要遵守程序员所期望的数据传播方式。在这个例子中，程序员所期望的过程是这样的：T1 先更改 A，之后更改 FLAG，T2 观察到（或者是可以读取）的这两次更改的顺序也必须是一样的，一旦 T2 返回 FLAG 的值是 0，那么 T2 就不能执行打印语句，而一旦 T2 返回 FLAG 值是 1，那么 A 的读数（打印的值）也必须是 1 而不能返回 0。存储模型保证了由一个线程产生的值传播到达另一个线程的合法或者允许顺序。

图 7-1 所示的通信模型表明内存地址新产生的值最终必须传播到所有的线程，这样这个新值才能被后续的 load 操作观察到。对于一份数据存在多份副本的情况，这意味着这个 store 的数据必须最终传播到所有的备份上，以便被其他线程读取到。这种特性通过 cache 一致性协议来保证，当出现 store 时，一致性协议就将对应的无效或者更新消息传播出去。cache 一致性在共享内存系统中提供了一种假象，它使得从软件看来，每个内存位置都只有一份单一的副本，尽管实际在物理上可能存在多个副本。

为了进一步说明这种特性，考虑图 7-2 中的带有 4 个迭代变量的 Jacobi 迭代算法的流程图。在 Jacobi 迭代中，每个迭代变量使用上一次迭代的值进行更新。为了实现这一功能，迭代变量的两个副本被保存在内存的两个向量 X 和 Y 中。在图 7-2 中，4 个线程每个更新 4 个迭代变量中的一个。首先， X_i 的新值根据 Y_i 的线性函数计算得到，然后 4 个线程进行 barrier 同步。barrier 同步要求必须等到所有线程都到达这个同步点之后，线程才能继续执行。然后，这 4 个线程根据 X_i 的线性函数重新计算 Y_i 的新值。如此反复迭代，直到根据某些标准，算法达到收敛。

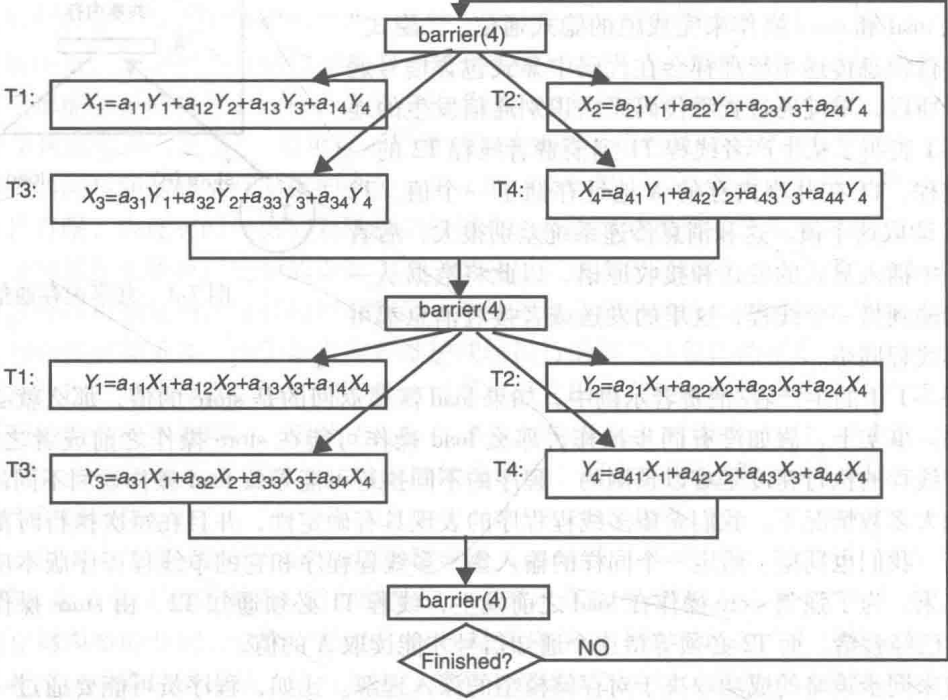


图 7-2 含有 4 个迭代的多线程 Jacobi 流程图

在这个算法中，所有的 X_i 新值必须在每次迭代的第一阶段和第二阶段之间传播给所有的线程。`barrier` 同步操作确保在迭代的第二阶段开始之前所有线程都已经计算出了 X_i 的新值，这样 `barrier` 保证在任何线程读取 X_i 时，都可以获得新值。

不管 X_i 或者 Y_i 是否存在多份副本，`barrier` 同步都是必需的。然而，如果存在多份副本（比如在多 cache 系统中），那么就需要一种机制来保证被线程 T 计算出来的 X_i 新值能传播到所有其他线程的 cache 中。传播过程可能发生在迭代的第一阶段，也就是值刚刚计算出来的时候立即传播，也可能发生迭代的第二阶段，也就是推迟到要读取这个值的时候。这时可能需要使用 `barrier` 来强制新值的传播。假设 X_i 新值在迭代的第二阶段读取，此时程序已经无法检测到 X_i 是否存在多份副本了，但即使有时候数据多个副本的值不相同，其执行结果仍然是满足一致性的。

这个简单的例子说明，一致性的保证并不要求 `store` 值立即传播，也不要求对应内存位置的所有副本必须在任何时候都一样。

7.2.2 硬件组件

在第 3、4 和 6 章中我们详细阐述了处理器、存储系统和互连网络的内容，在本节中，我们会简单回顾这些硬件组件的概念，以便更好地理解本章的内容。在现代多处理器系统中，为了提高硬件效率，解决诸如存储墙这样的问题，硬件组件也变得越来越复杂。这些组件包括处理器、cache 以及互连结构等，下面我们将给出这些组件的一些基本描述。

处理器

首先考虑简单的五级流水顺序处理器，带推测执行的乱序处理器将在 7.7 节中进行介绍。图 7-3 所示的是带有 `store` 缓冲和写回 cache 的五级流水线，`store` 指令在到达访存（ME）流水级并通过 TLB 检测之后就不会发生异常，可以直接执行和提交。当 `store` 指令等待更新 cache 时，会被插入 `store` 缓冲中，之后，依据 `store` 缓冲的管理策略，`store` 指令会在 cache 中完成执行。所以只要 `store` 缓冲没有满，在 ME 阶段的 `store` 指令就是非阻塞的（non-blocking）。“非阻塞”意味着 `store` 指令可以在一个周期内通过 ME 阶段，而不会阻塞流水线。`load` 指令是阻塞的，这意味着在 cache 可以返回读取值之前，处理器会被暂停。`store` 缓冲中的 `store` 指令会和 `load` 指令竞争 cache 的访问权，如果 `load` 地址的值和 `store` 缓冲中某个 `store` 操作的地址匹配，那么可以直接从 `store` 缓冲中返回对应的 `load` 值。

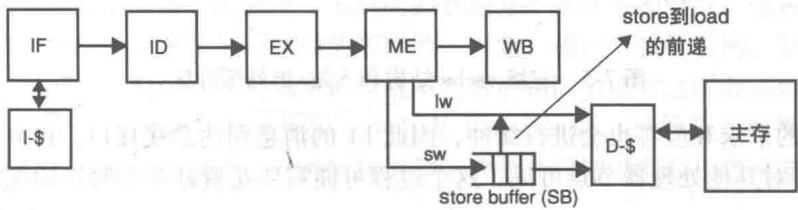


图 7-3 带有 `store` 缓冲和写回 cache 结构的五级顺序流水线

由于 `store` 缓冲（几乎所有带流水的机器都会配备）的存在，`load` 和 `store` 指令可能在 cache 中乱序执行，目前的单处理器系统都广泛采用了这一基本优化策略。然而在多处理器系统中，如果支持 `store` 缓冲，那么一致性和存储模型都将更加难以实现。

cache

现代处理器中的 cache 通常是非阻塞的，或者也称为无锁的，因此可以支持图 7-3 所示结构中 `store` 操作那样的非阻塞内存访问。无锁 cache 利用了 cache 具有两个接口的特征：一个连

接处理器, 另一个连接下一级 cache 或者内存, 如图 7-4 所示。当发生 cache 失效时, 在处理器端的控制器不会阻塞处理器, 相反这个失效请求会提交给在内存端的控制器, 同时在处理器端可以继续发出更多的访问操作。待处理的失效都存储在特殊寄存器中, 当失效的数据块返回时, 处理器所需的值就会进行快速前递。

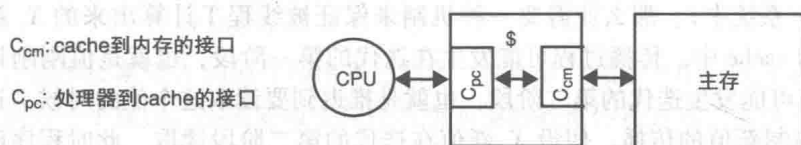


图 7-4 无锁 cache

因此, 在无锁 cache 结构中, 多个 cache 失效操作可以和 cache 命中操作并发处理, 失效的开销可以通过其他 CPU 操作以及其他失效操作重叠隐藏。在图 7-3 中的无锁 cache 结构下, 多个 store 失效以及最多一个 load 失效/命中请求可以同时处理, 因为 load 操作在处理器中是阻塞的, 而 store 是非阻塞的。

为了降低平均的 cache 失效开销, CPU 节点的 cache 空间通常被组织成多个层次, 如图 7-5 所示。通常情况下, 不同层次之间的数据采用包含的关系, 这样那些不影响更高层次 cache 的输入消息就可以过滤掉。入站请求和应答会经过多级缓冲向上渗透到 L1 cache, 设置多级缓冲的目的是为了均衡不同 cache 层次之间的处理流量。在这种机制下, 入站消息到达网络或总线接口后, 可能需要经过很多个时钟周期才能最终达到 L1 cache, 然后处理器才能够从 L1 中读到该消息对应的数据和指令。在这段时间内, 处理器完全不知道这个入站消息的存在, 它会继续处理其他操作并通过多级缓冲往外发送出站消息。这些延迟的存在导致了处理器的多个访存事务之间存在大量的时间重叠, 此外, 每个处理器节点上的延迟也是不同的, 因为每个处理器核以及 cache 上的执行时间和访存事件都可能各不相同。

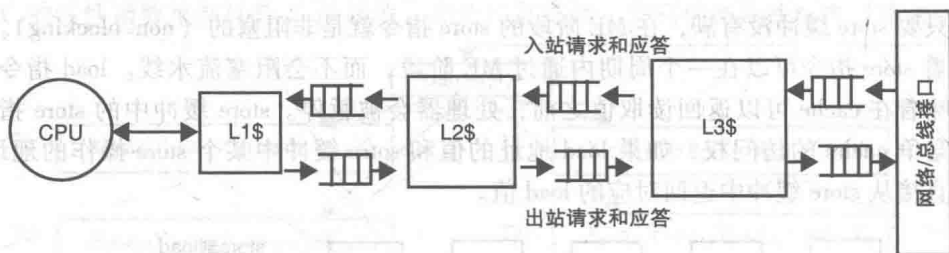


图 7-5 三级 cache 结构和入站/出站缓冲区

从 L1 出站的请求和应答也会进行缓冲, 因此 L1 的消息到达总线接口, 再发送到总线或者互连网络, 最终对其他处理器节点可见, 这个过程可能需要花费好多个时钟周期的时间。

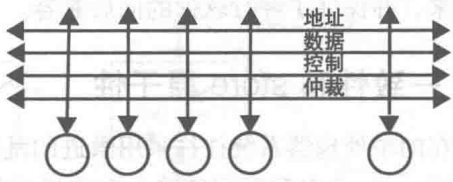
系统互连

系统互连是处理器和内存之间消息交换的传输媒介, 最简单的系统互连结构是总线。

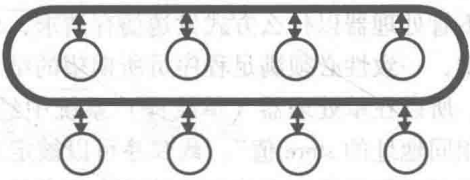
总线是一种被动形式的连线, 它可以高效地维护 cache 一致性, 因为总线可以在节点之间进行高效的广播 (broadcast) 和总呼 (broadcast)。广播是指数据从一个处理器节点传输到其他所有处理器节点; 总呼是指广播一个命令到所有节点, 然后所有节点都进行应答。此外, 总线可以对无法在 cache 中完成的访存操作进行串行排序。我们在后面将会看到, 这种特性有助于一致性和存储模型的实现。然而, 总线的带宽有限, 并且非常容易达到饱和, 因为所有的总线事务流量 (即使是不相关的流量) 都需要占用同一电线进行传输。此外, 总线的传输速度也

受限于总线的长度和连接到它上面的所有电容负载。

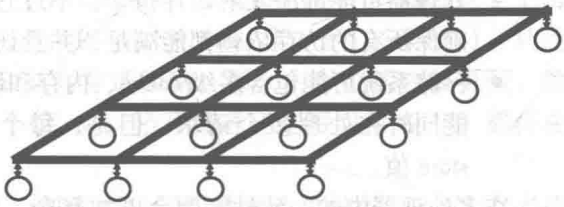
系统总线通常由多根总线组成，如图 7-6a 所示的仲裁、地址、数据、控制总线等。总线必须首先通过集中式的或者分布式的仲裁机制进行仲裁。一旦获得总线控制权，就可以发出地址信号（如果是 store 操作则还有数据信号）和控制信号，然后就可以接收到数据信号（load 的情况）或者确认信号（store 的情况）。这些步骤可以流水化以实现更大的吞吐量。为了进一步提高带宽，总线可以采用数据包交换（packet-switched，或者叫分片事务（split-transaction））机制。在电路交换机制下，在整个事务的处理过程中总线都是独占的。而在包交换（分片事务）机制下，一旦请求（地址 + 控制）信号被发送，总线就可以释放，当应答信号（数据或者确认）就绪之后，应答端重新仲裁总线请求，获取总线后应答以一个新的数据包的形式发送。在请求和应答数据包之间，总线是释放的，所以多个总线事务可能同时在处理，这些并发的总线事务一般访问不同的地址。



a) 总线



b) 环



c) 二维mesh

图 7-6 系统互连示例

在环或者 mesh（见图 7-6b 和 c）这样的点对点互连结构中，多个访存事务可以同时处理，只要它们在同一时钟周期不会使用相同的网络链路即可。在这种网络中，每个处理器位于互连结构的一个节点上，节点之间通过一定的路由策略沿着对应链路将消息传输到目的地。点对点消息可以使用的带宽比总线要大得多，因为点对点消息可以使用不同的无冲突路由节点。然而广播或者总呼的带宽是固定的，因为所有的节点都必须通知到。分布式互连的另一个优点是点对点链路的时钟频率可以比总线更快。

在分布式网络中，两个节点之间可能存在多个路由器，自适应网络可以根据网络中的拥塞情况动态改变消息的路由策略。因此两个节点间的延迟并不总是可预测的。这种网络无法为所有请求和应答消息提供一个全系统范围内的顺序性，因此，相比于总线结构，这类网络结构上的存储模型更难实现。有时候，这类网络甚至都不会维护相同节点对之间数据包的发送和接收顺序，当消息从一个节点（源）发送到另一个节点（目的）时，目的节点的接收顺序可能与源节点的发送顺序不一致，很显然，这种网络上的一致性和存储模型都比较难实现。

存储模型需求

复杂微处理器结构、cache 层次结构、片上/系统互连等诸多因素的差异，使得多核/多处理器发出的访存请求的延迟和顺序难以确定，而这种复杂性又不能直接暴露给程序员，主要有两个原因：（1）如果暴露给程序员去保证，那程序的正确开发、编译、移植将会变得极其复杂；（2）无法实现软件的向后兼容，这导致我们不得不回到为每台新机器重新开发软件的原始时代。

因此，既然我们无法把每一个体系结构组件的特征都暴露给程序员，那么体系结构设计人员就必须为程序员提供一个合适的模型，即体系结构和软件设计人员需要在硬件效率和可编程性之间找到一个折中，这个折中的结果就是存储模型。存储模型提供了一系列简单的规则，规

定了多处理器中的访存操作行为。作为 ISA 规范的一部分,存储模型使程序员从硬件复杂度中解放出来,并保证了并行软件的向后兼容。

7.3 一致性和 store 原子性

现在的单处理器系统往往采用激进的乱序执行访存操作,存储系统结构复杂、层次众多,支持无锁 cache 和各种写回策略,在这样的复杂结构下,存储一致性是一个非常麻烦的问题。好在单线程上下文中已经强加了一个严格定义的线程序,这才使得上述问题变得相对简单一些。不管处理器以什么方式发送访存请求,也不管内存系统以什么方式进行应答,在单线程上下文中,一致性必须满足程序员所期望的结果:指令看起来像是严格按照线程序每次执行一条指令。所以在单处理器(单线程)系统中经典的一致性定义是“load 必须以线程序返回最近一次相同地址的 store 值”。线程序可以被定义为取指或译码的顺序,或者一次只执行一条指令时的执行顺序。这一定义意味着两个硬件组件——处理器和内存系统需要进行协作。

- 处理器可能乱序发射访存指令,不过它会通过访存相关硬件(比如 load/store 队列)来确保所有的访存依赖都能满足,并且访存操作看起来就好像按照线程序执行一样。
- 存储系统可能包含多级 cache、内存和硬盘,每一级都可能有不同的写回策略,并且可能同时在处理多个请求,但是,每个 load 操作最后必须以线程序返回最新的一个 store 值。

在多处理器中的一致性会更加复杂,因为没有类似“多线程序”这种概念。要想让上文对一致性的经典定义(包含“最近的 store”的概念)继续适用,不同线程之间对同一地址的所有访存操作之间就必须存在一个全局的时间序。

在这一章中,我们假设单线程的一致性或者线程内部的访存依赖关系都是满足的,而在其他情况下,比如不同线程之间的访存操作顺序,则允许放松一下。单线程必须确保正确执行,因此上面的这种约定是必要的,在下文将不再重复说明。

7.3.1 多处理器一致性的实现困难

在传统的单处理器系统中,当 I/O 操作绕过在系统总线上的 cache,而直接流入/流出内存时,也会存在 I/O 一致性问题。但是这种问题可以通过软件的方法来解决,因为 I/O 事件不会频繁发生,并且 I/O 造成的中断和陷阱是软件可感知到的。传统的解决办法包括:设定不可缓存的地址空间,设置不可缓存的内存操作,以及进行 cache 刷新等。

在多处理器中,一致性问题无处不在,出于性能的考虑,共享内存系统中的通信一般都是隐式完成的,潜在的问题也极少会显式通知软件。在多处理器系统中的一致性问题是由于对同一个内存地址的多份副本造成的,这里的多份副本不仅仅是在 cache 中,也包括处理器核中和访存相关的一些非常隐蔽的底层硬件缓冲区,比如图 7-3 中的 store 缓冲区。

图 7-7 展示了在没有任何硬件一致性支持下的 cache 一致性问题,图中 3 个处理器访问同一个内存地址 X,每次访问称为一个内存事件。首先,假设图中描绘的这 5 个内存事件都立刻、原子地、零时间内执行,或者假设这 5 个访存操作完全没有时间上的重叠。为了更好地解释其中的含义,定义访存指令的生命周期,这个周期从指令取指时间开始,到该访存指令在整个系统中的所有活动都已经完成为止。很明显,在指令的生命周期之外,访存操作不会影响到其他执行的输出结果。假如两个访存指令的生命周期不相交,那么它们在时间上就不会重叠,但即使是这种情况,一致性问题也不是那么容易理解。

假设所有的访存事件在时间上是不重叠的,让我们跟踪这些事件的进展。最初, X 的单一

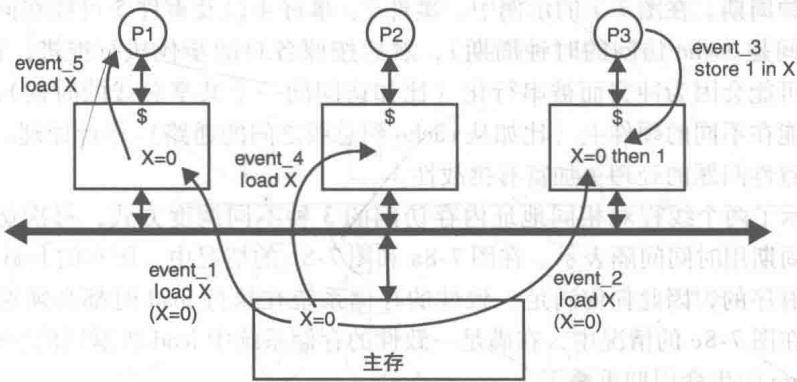


图 7-7 不带 cache 一致性协议的 cache 一致性问题

副本存在内存中，因此这个执行默认是一致的。在事件 1 和事件 2 的 load 操作之后，X 同时存在 P1 和 P3 的 cache 中，因此形成了 3 份副本，2 个在 cache 中，1 个在内存中。因为这 3 个副本的值相同（都是 0），所以执行仍然是一致的。然后 P3 在 X 中 store 了 1 个 1（事件 3）。从此时开始，系统中有 3 个副本和 2 个值。假如 cache 是写穿透策略，那么内存也随之被更新，然后 P2 的事件 4 读到 X = 1，这也是对的。然而，不管 cache 是写穿透还是写回策略，P1 都将在事件 5 中读取到 X = 0。

讨论一致性问题的时候，如果满足以下条件，则认为一致性是自动维护的：（1）在当前系统中只存在一份副本；（2）存在多份副本，但是多份副本有相同的值。很明显，如果一个值的所有副本在任何时候都是相同的，那么就没有必要考虑一致性的问题了。在图 7-7 中，在事件 3 之后，一共存在 3 个副本，在系统范围内有两个不同的值，然而这并不意味着执行过程是不一致的。除非副本中的不同值会对软件产生影响，否则这种不同就无关紧要。为了观察到违背一致性的情况，线程必须执行后面的 load 操作，这样才能观察到不同的值。因此，即使在事件 3 之后，整个执行过程仍然是一致的。

另一个有趣的问题是，如果在事件 4 或者事件 5 的执行过程中，P1 或者 P2 返回 0 的话，是否还满足一致性？假如 P1 和 P2 的 load 操作和 P3 的 store 操作在时间上没有重叠或者是零延迟的原子执行，那么访存事件就完全可以按照时间进行排序，一致性经典定义中“最近的副本”的概念显示事件 4 和 5 必须返回数值 1。所以假如访存事件在时间上没有重叠或者原子执行，那么任何一致性协议都必须保证在事件 4 和 5 中返回值 1。

我们知道，在编写并行软件时通常认为软件行为是独立于具体的实际时间的，如果软件检测不到上面所讲的这种明显的不一致现象，那么这种不一致就无关紧要。在当前环境中，P1 或者 P2 在满足一致性的情况下也不一定返回 1，除非它们的访存操作在时间上没有重叠或者是原子执行的。图 7-1 说明了其中的原因：P3 的执行可能会更慢，或者 P1 和 P2 的执行可能更快（或者同时出现），在这种情况下，P3 的 store 可能在真实时间里发生在 P1 和 P2 的两个 load 操作之后。

如果程序员期望得到确定的结果，那么 P3 的 store 操作和 P2、P1 的 load 操作应该用一个同步事件（比如 barrier 同步）分隔开，因为正如图 7-1 所示，共享内存系统在硬件上不会对不同线程对相同地址的 load 和 store 操作进行定序。在这种情况下，同步操作强制了 store 和两个 load 操作的先后顺序，因此 P1 和 P2 的两个 load 操作一定会返回 1。

在真实机器中，访存事件并不是瞬时完成的，通常情况下不同访存的执行过程会在时间上有重叠，在整个生命周期内，每个访存指令在硬件上都要经过一串很长的处理过程，这可能会

花费大量的时钟周期。在图 7-7 的示例中，事件 3、事件 4 以及事件 5 可能在同一个时钟周期触发（触发时间是 cache 访问的时钟周期），然后按照各自的步伐执行推进，在处理过程中，这些事件有时可能会因为冲突而被串行化（比如访问同一个共享总线的时候），在其他时候，不同事件又可能在不同的硬件上（比如从 cache 到总线之间的通路）并行处理。这种潜在的时间重叠使得一致性问题的处理更加富有挑战性。

图 7-8 显示了两个线程对相同地址内存访问的 3 种不同调度方式，每次访问（load 或者 store）的生命周期用时间间隔表示。在图 7-8a 和图 7-8b 的情况中，所有的 load 操作和 store 操作之间是时间有序的，因此任何满足一致性的存储系统在执行 load 时都必须返回前一个 store 的值。然而，在图 7-8c 的情况中，在满足一致性的存储系统中 load 所返回的值也是不确定的，因为 load 和 store 的生命周期重叠了。

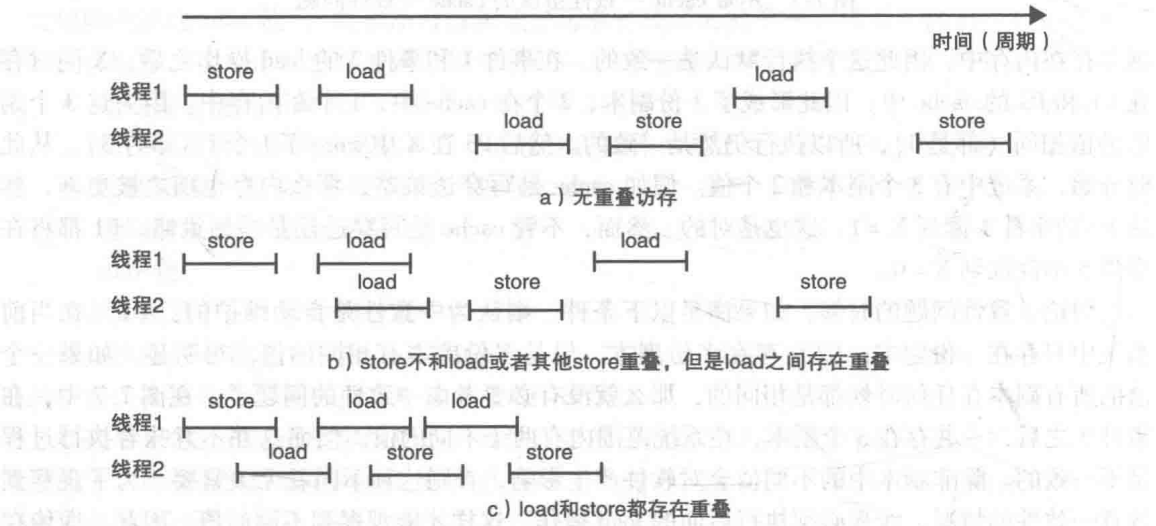


图 7-8 两线程对相同地址访存操作的时间重叠

当 load 和 store 有时间重叠时（如图 7-8c 所示），图 7-7 中的关键是理解硬件上是如何处理的。不过，我们先不考虑这个关键问题，下面先介绍一些简单的协议和系统，这些内容我们在本章中会多次用到。

7.3.2 cache 协议

如果假定访存操作瞬时完成（原子性），或者不同访存操作在时间上完全没有重叠，那么协议可以表示为简单的状态图表。这种表示 cache 稳定状态之间转换规则的定义叫作协议的行为规范。有时候，协议可能有临时状态，这种瞬时状态出现在从一个稳定状态到另一个稳定状态过渡的时候。不过瞬时状态一般只是概念上存在，往往用来在模块内部记录全局状态的转变过程，因此，在本章中我们忽略这种瞬时状态，这不会对接下来的讨论产生影响。

在第 5 章中，我们已经深入讨论了 cache 协议以及各种系统所需要的硬件结构，在本章，我们主要讨论总线和简单 cc-NUMA 多处理器上的侦听协议，如图 7-9 所示。我们只讨论两种简单的 cache 协议——MSI-无效和 MSI-更新，更加复杂的协议难以描述，而且不易理解，在这里暂不介绍。

侦听协议

简单的多 cache 系统可以基于总线互连结构所提供的广播机制来实现，如图 7-9a 所示。总

线接口 (BI) “侦听”在总线上的控制和地址广播消息，如果当前的总线事务会影响到本地 cache 的内容或者需要进行响应，那么本地总线接口会把这个消息转发给本地 cache，否则就直接丢弃。

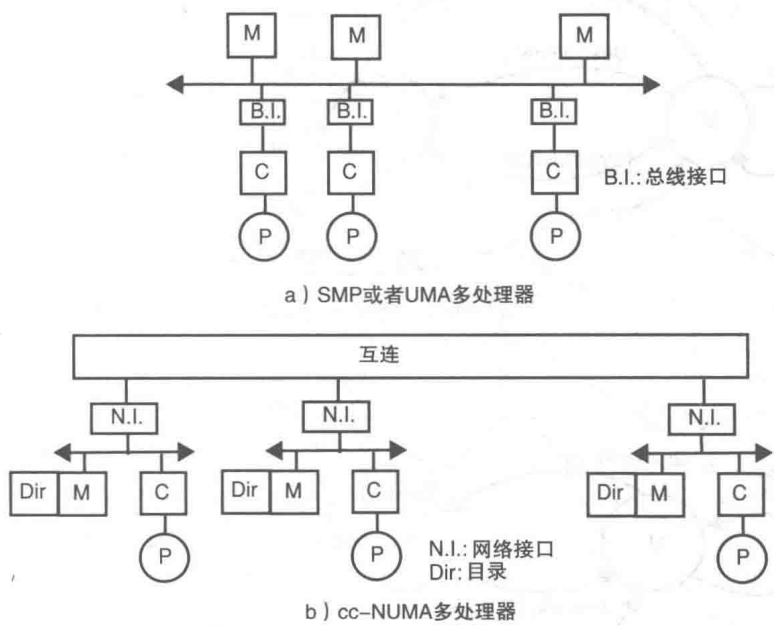


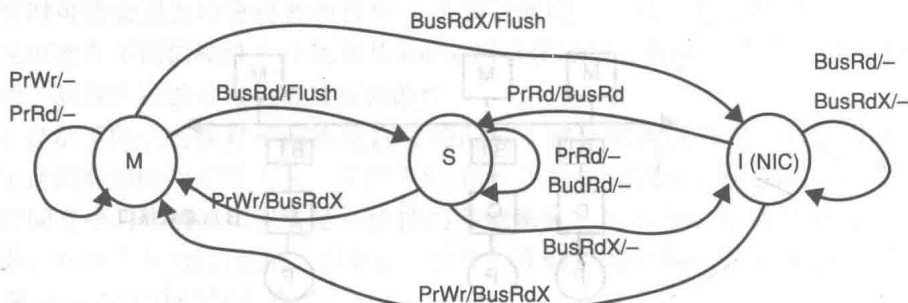
图 7-9 基于 cache 的多处理器配置

在行为级别，基于侦听的一致性协议可以大致分为“写 - 无效”和“写 - 更新”这两种协议。通常来说，协议根据其稳定状态的起始字母来命名。在这一章中，我们为写回 cache 提供两种协议，这两种协议都是基于三个本地 cache 状态：修改状态 (modified, M)，共享状态 (shared, S) 和无效状态 (invalid, I)。无效意味着对应的数据块在 cache 中，但是已经无效了，所以它的内容是旧的或者没有被缓存 (Not In Cache, NIC)，我们将这两个协议分别称为“MSI-无效”和“MSI-更新”。

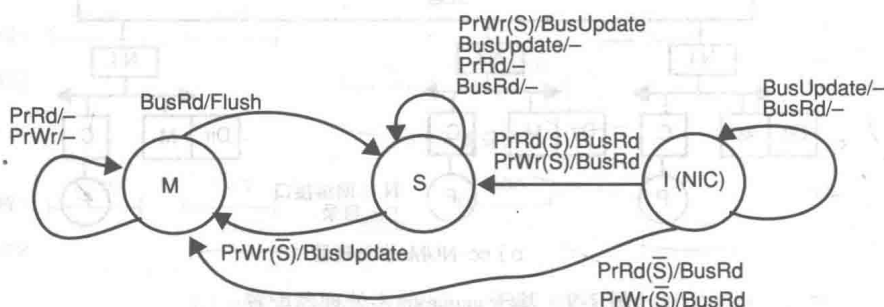
在修改 (M) 状态时，本地的数据副本是整个系统中唯一有效的，而内存中的数据是旧的，本地处理器可以从 cache 行中随意读写该数据。MSI-无效和 MSI-更新之间的关键不同在于共享状态 (S)。在共享状态下，系统中可能存在多个副本，必须维护一致性，此时两种协议下的内存数据都是最新的。在 MSI-无效协议中，cache 中有共享副本的处理器不能更改这个副本，而只能读。要想更改这个数据，cache 必须先获得修改 (M) 状态的副本。为此 cache 必须无效掉所有的远程副本。与之相反，在 MSI-更新协议中，处理器可以任意地读取或者更改它的共享副本，不过，每次 store 操作引起的更新必须广播到所有的远程副本中，以保持数据的一致性。每次发送更新请求时，总线上的共享线路会反馈是否存在远程副本 (使用 broadcast 实现)，当没有 cache 响应时，本地 cache 切换到修改状态 (M)。需要特别强调的是，在更新协议中没有“无效”请求这种类型。在共享状态下 cache 可以通过共享线路检测自己是否具有唯一副本。在 MSI-更新协议中的共享状态有时候也叫作写穿透状态。

图 7-10 的状态图描述了对应的协议机，即每个内存块对应 cache 的有限状态机行为。系统中的所有内存块和 cache 都会对应到一个这样的状态图。这个协议机本质上是个 Moore 有限状态机，输入来自于处理器和侦听总线。来自处理器的输入是 PrRd (处理器读) 和 PrWr (处理器写)。在 MSI-无效协议中，来自总线的输入是 BusRd (读操作 miss) 和 BusRdX (写操作 miss)；在 MSI-更新协议中，来自总线的请求是 BusRd (读或写 miss) 或者 BusUpdate (数据写穿透)。针

对上述输入, MSI-无效协议的本地协议机使用刷新操作(传递副本值并更新内存)、BusRd 或者 BusRdX 进行响应。在 MSI-更新协议中, 响应方式则是刷新、BusRd 或者 BusUpdate。



a) MSI-无效协议的协议机



b) MSI-更新协议的协议机

图 7-10

当内存块在 cache 中处于 S 状态的时候, 在 MSI-无效协议中, PrWr 发送一个 BusRdX, 而在 MSI-更新协议中, PrWr 发送带有地址和数据的更新请求。注意在 MSI-更新协议机中使用的信号 S 是一个二进制标志位(flag), 用于标识总线共享线路上的应答, 并且是协议机的一个输入, 但它是协议机的后期输入, 出现在总线请求之后, 并且决定下一个状态。

图中这两个协议被简化到了最低程度。比如, 在 MSI-无效协议中, 其实我们也可以在 PRWr 上发送一个新的 BusUpgrade 请求到共享副本, 而不是发送 BusRdX 请求, 后者需要重新装载数据块, 而这个 BusUpgrade 请求只是简单地把远程 cache 副本无效掉。

同样, 在 MSI-更新协议中, 当处理器执行了一个 store 操作时, 从 I 到 S 的转换会在 BusRd 请求上发送一个更新操作。但如果考虑这点, 那么这里的总线协议又将变得非常复杂, 以致难以讨论下去。因此, 如图 7-10 所示, 不管是 MSI-无效协议机还是 MSI-更新协议机, 我们都假设当访存操作在 cache 中无法完成时, 都会由处理器进行重试。比如, 当处理器执行 store 操作, 并且总线中的共享线路为高电平时, MSI-更新协议分两步完成从 I 到 S 的转换: 首先, 状态 I 的 cache 发送一个 BusRd 信号, 它会读取一个共享备份到 cache 中; 然后, 处理器重试这个 store 操作, 此时, 数据副本是 S 状态的, 所以会发出一个 BusUpdate 信号, 然后处理器成功提交这个 store 指令。

目录协议(cc-NUMA 系统)

上面介绍的 MSI-无效协议和 MSI-更新协议都是在基于侦听总线的多处理器环境下设计的, 它们同样可以扩展到图 7-9b 所示的 cc-NUMA 系统中。这类系统通过基于目录的点对点消息来保证一致性, 在目录中记录了所有内存块的全局状态。每个内存块可以通过其地址找到对应的

主节点，每个内存块在其主节点上都有一个对应的目录项，该目录项中包含了指向所有数据副本的指针以及表示该数据块全局状态的状态位。此外，在主节点上，每个目录项都有一个忙（busy）位，它指示当前数据块是否有事务正在处理中。如果存在未完成的事务处理，那么控制器将不会对任何其他针对该数据块的请求进行应答，因此其他请求将被拒绝，之后再重试。

- 通常情况下，参与一致性事务的节点包括：
- 主节点（H）：内存块及其目录项所在的节点。
 - 请求节点（R）：产生请求的节点。
 - 脏节点（D）：包含最新（修改过的）数据副本的节点。
 - 共享节点（S）：包含数据共享副本的节点。

下面举一个协议事务处理的例子，在 MSI-无效协议中发生了一个 store 失效，此时在其他节点上存在两个共享副本。图 7-11a 给出了在这个事务处理中的点对点消息交换。请求节点发送一个 BusRdX 消息到主节点。主节点搜索对应的目录项，找到后通过置 busy 位将其锁定，然后向两个共享副本发送令其无效的消息。该无效请求经确认后返回到主节点，然后主节点将数据块发送给请求者，并重置 busy 位以解除目录项的锁定。图 7-11b 给出了在 MSI-更新协议中，当共享节点上出现 store 命中时的点对点消息发送情况。请求节点先发出 BusUpdate 请求到主节点，然后主节点更新所有远程共享副本（而不是无效掉），最后向请求节点进行应答，确认事务结束。

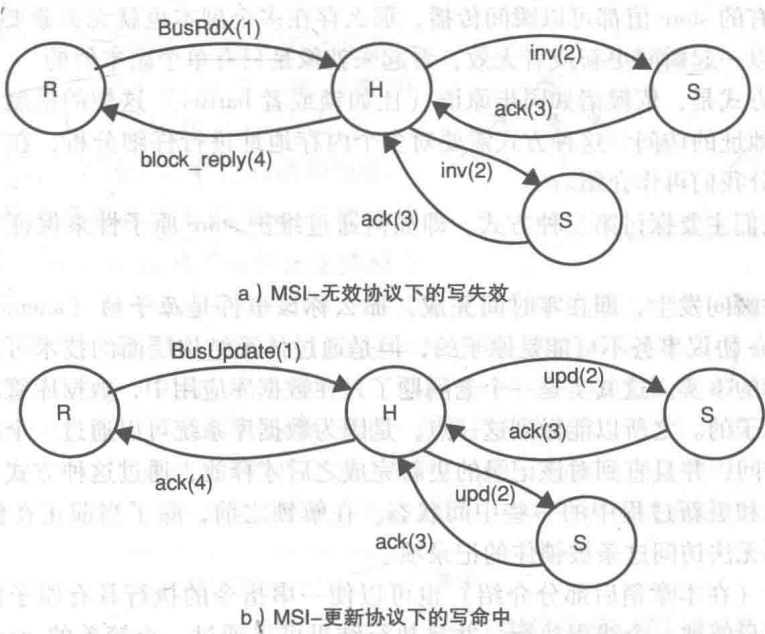


图 7-11 MSI 协议下存在两个共享副本的写访问执行情况

7.3.3 store 原子性

数据一致性的作用在于即使相同内存位置对应有多份数据副本，但是对于软件来说都好像是只有一个副本似的。数据一致性有多种实现方法，比如，避免相同内存位置出现多个副本，或者将 store 的值立即（零延迟）更新给所有副本，就好像 store 操作是原子完成的。后面这种情况通常叫作 store 原子性。显然，假如 store 操作可以瞬间影响所有副本（在 MSI 协议中，要么是快速无效掉其他 cache，要么是快速更新其他 cache），那么一致性就能保证，因为在这种情况下，所有多个副本本质上就和单个副本一样。不过，现代处理器系统都需要 cache，因此

不可避免会出现多份副本的情况；此外，复杂的 cache 事务处理也不可能瞬间完成。

我们从第一个公开定义的多处理器一致性协议开始介绍，这个一致性定义叫作“严格一致性”。

定义 7.1 (严格一致性) 如果存储系统中的 load 指令总是能返回相同地址最近一个 store 的值，那么称其满足严格一致性。

这种定义是从单线程单处理器环境中借鉴来的，这种环境中存在明确的线程序，而在多线程系统中却难以使用，因为不同线程执行的对相同地址的 load 和 store 操作的顺序是没有明确定义的。在多线程环境下，线程的相对执行速度是不断变化且难以预测的，而且处理器核之间也不存在能够将线程的 load 和 store 信息瞬间完成共享的所谓“超高速”链路。

要想继续使用严格一致性的定义，必须对所有相同地址的 store 操作实现全局的时间序，下面四种方式可以实现这样的全局序：

- 第一种方式是，所有内存地址都只维护单个副本，但在现代微处理器中，cache 和数据多个备份基本上是必需的，因此这种方法实际意义不大。
- 第二种方式是，如果对相同地址的内存访问在时间上没有重叠（或者起码 store 操作不和 load 或者其他的 store 重叠，参见图 7-8），那么也存在一个全局的时间序。但这一要求在实际系统上也基本上不存在，因此也无法使用。
- 第三种方式是，同一地址的 store 操作结果可以瞬间传播到系统范围内的所有副本上。假如所有的 store 值都可以瞬间传播，那么存在多个副本也就无关紧要，因为所有的副本都可以一起瞬间更新或者无效，看起来就像是只有单个副本似的。
- 第四种方式是，依赖诸如同步原语（比如锁或者 barrier）这样的机制来强制隔离对相同内存地址的访问。这种方式需要对多个内存地址进行仔细分析，在本章后面的存储模型部分我们再作介绍。

在本节中我们主要探讨第三种方式，即如何通过维护 store 原子性来保证存储系统中的严格一致性。

若一个事件瞬间发生，即在零时间完成，那么称该事件是原子的（atomic）。虽然实际物理系统中的 cache 协议事务不可能是原子的，但是通过体系结构层面的技术可以对软件隐藏硬件不满足原子性的事实。这其实是一个老问题了，在数据库应用中，数据库管理软件所看到的记录更新都是原子的。之所以能做到这一点，是因为数据库系统可以通过一个特殊操作来锁定对某条记录的访问，并且直到对该记录的更新完成之后才释放，通过这种方式，数据库系统隐藏了在记录读取和更新过程中的一些中间状态。在解锁之前，除了当前正在修改它的线程之外，其他线程都无法访问这条被锁住的记录项。

临界区同步（在本章稍后部分介绍）也可以使一串指令的执行具有原子性。临界区内的代码在同一时刻只能被一个线程执行，并且执行结果可以通过一个简单的 store 或者解锁操作原子地（至少在软件层面看起来是原子的）释放给其他所有的线程。

因此，也可以认为“原子性”就意味着“不可分割”，这也是这个词最原始的含义。“不可分割”在本文中意味着，当一个复杂事件发生时，在整个事件完成之前，没有哪个线程可以观察到这个事件引起的中间变化过程，而一旦事件完成，其造成的影响就对所有线程可见。在物理系统中，这是实现复杂操作原子性的唯一可行方法。

定义 7.2 (store 原子性) 如果不同线程永远无法同时观察到同一内存位置的不同值，那么这个 store 就是满足原子性的。

在上述定义中，对应每个内存地址，在任何时刻都只能有一个可读的值。同一份数据可能在不同的存储结构中有不同的值，但是只有一份是可以通过 load 读取到的。这样内存位置的

值可以根据实际时间进行定序，对于任意一个给定的内存地址，它的值都可以形成一个全局时间序，因此，store 原子性保证了上面定义的严格一致性。

store 原子性是一个很严格的条件，当然，在现有的计算机体系结构中，存储系统往往暴露给软件一种假象，即 store 具有原子性，这种现象是完全可接受的。因此，定义 7.2 中的描述不应该被理解成一个必要条件，暂时我们把它当成一个充分条件。

基于总线系统的 store 原子性

图 7-12 给出了一个支持 cache 原子访问的理想总线结构，假设具有原子性的存储系统部分用粗线表示（在本章中，将用粗线来表示支持原子性的存储子系统）。处理器发起的访存操作只要能在本地 cache 中完成，那么就可以并发进行，然而，一旦处理器不能在本地 cache 完成访存操作，就需要同时在总线上以及所有 cache 中执行一个原子性的协议事务。

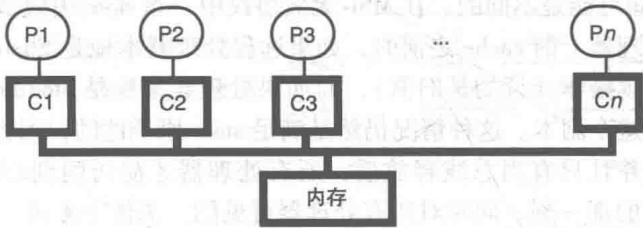


图 7-12 支持原子 cache 访问的 cache 系统

下面我们将引进一些形式化的方法以便进一步进行分析，在本章余下部分中，我们将用图 7-13 中的标记来表示 load 和 store 的执行。一旦 load 或 store 执行，就能知道对应的值和地址。



图 7-13 load 和 store 执行的标记方法

例 7.1 支持原子内存访问的 MSI-无效协议

下的严格一致性 图 7-14 说明了为什么支持原子协议事务（Atomic Protocol Transaction, APT）的 MSI-无效协议是满足严格一致性的，协议事务在图中用粗体进行了标识。最开始，cache C1 有一个 Modified 状态的副本，并且在本地完成了 load 和 store 操作。之后，cache C2 遇到一个读失效，引起了一个原子协议事务（APT1），进而导致在 C1 和 C2 上出现两个共享副本。这两个本地副本一直都只有读操作，直到 APT2 出现，此时 C2 请求一个 Modified 副本，并且通过原子操作完成。最后，APT3 完成后，C1 获得一个 Modified 副本。上述访问在时间上是有序的，协议保证了每个 load 操作以时间顺序返回最新的 store 值。

C1:	C2:	备注
$S^1(A)a_1$		INIT: 数据不在 C2, A 在 C1 中修改
$L^1(A)a_1$		
$S^1(A)a_2$		
$L^1(A)a_2$		
-----	$L^2(A)a_2$	APT1: C2 中读失效; A 在 C1 和 C2 中变为共享状态;
$L^1(A)a_2$		两个线程都可以读到 A=a2 的结果;
	$L^2(A)a_2$	线程都不允许写操作

	$S^2(A)a_3$	APT2: C1 中数据被无效, A 在 C2 中变成修改状态
$S^1(A)a_4$		APT3: C1 中发生写失效;
$S^1(A)a_5$		C2 中数据被无效掉

图 7-14 支持原子协议事务（MSI-无效协议）的执行

例 7.1 中的理想行为可以在真实物理系统中实现, 实际上, 在 20 世纪 80 年代早期, 多处理器系统中的处理器通过一根电路交换的总线结构进行连接, 处理器每次执行和提交一条指令, 没有使用 store 缓冲, 这有点类似于基本的五级流水线。而一致性协议采用的是基于无效的方式。当碰到访存指令, 且 cache 状态需要协议介入时, 处理器就会阻塞住, 直到访存操作完成。为了处理该访存操作, cache 控制器需要先获取总线的使用权, 并且直到整个协议事务完成之后才进行释放。因此, 任何时候只能同时有一个一致性事务被处理。在这种系统中, 一致性事务的处理在时间上没有任何重叠, 因此, 协议处理过程严格按照有限状态机的规范进行转换, 类似于图 7-12 中所描绘的具有瞬间完成、原子性的协议事务处理功能的理想系统。

有人可能会对上面最后一句的说法提出疑义, 因为电信号通过线路的时候是存在延迟的。在实际物理系统中, 由于 store 失效触发的总线和协议事务必须通过总线进行传播, 因此到达不同远程 cache 的时间可能是不同的。在 MSI-无效协议中, 当 store 失效请求发送到总线上, 但是还没有到达远程处理器上的 cache 之前时, 如果远程处理器本地是 Shared 状态的副本, 那么就可以一直对它进行读操作 (读的是旧值), 而如果处理器本地是 Modified 状态的副本, 那么就可以一直读或者写这个副本。这种情况仍然是满足 store 原子性的, 因为远程 cache 的副本最终会被请求无效掉, 并且只有当总线释放后, 所有处理器才能访问到新值。也就是说, “新值”是在总线被释放的那一刻, 同时对所有处理器可见的。总体上来讲, 每个存储地址的数据都会交替经历两个不同的阶段: 一个阶段是存在多个副本, 并且多个副本是一致的、只读的; 另一个阶段是只存在一个副本, 并且这个副本是可修改的 (同样满足一致性)。这两个阶段之间的转换是原子性的, 在任何时刻只有一个值能被访问到。

需注意的是, 原子协议事务也同样可以在基于分离事务的总线结构上实现, 前提是总线协议能够保证, 针对同一个存储块同时最多只能在总线上发起一个协议事务。如果有多条总线, 那么必须确保相同存储块的事务总是使用同一条总线进行处理。

定义 7.2 没有给出实现 store 原子性的通用方法。接下来我们根据定义 7.2 推导出确保 store 原子性的一个简单充分条件, 在推导过程中, 会引入几个有用的定义。

访存原子性的充分条件

通常情况下, 我们认为 load 操作是原子性的, 因为 load 一旦获取到对应的 store 值后, 这个值就不会再更改, 一直到 load 指令提交。另一方面, store 操作则需要一定的时间才能将值传播到整个系统中。下面先介绍一些在本章中会多次用到的定义。

定义 7.3 (访存完成)

- 当线程 i 的 load 指令无法获取到 store 操作之前的值时, 称该 store 操作相对于线程 i 执行完成。
- 当 store 操作对所有线程都已经完成时, 称 store 操作全局完成。
- 当 load 的值已经锁定并且不能被取消时, 称 load 操作完成。
- 当 load 已经完成, 并且产生该 load 值的 store 操作也已经全局完成时, 称该 load 全局完成。

根据存储系统和处理器的执行规则, load 的值一旦锁定, 那么该 load 操作就已经完成。具体来讲, load 操作完成时, 它返回的可能是邻近线程中的任何有效值 (任何符合一致性的值), 而处理器会屏蔽掉所有可能影响这个返回值的事件, 并且将 load 的值提交到寄存器中。

在图 7-3 的顺序处理器架构中, 当 load 的值被锁定之前, 执行无法继续推进, 因为线程需要这个 load 的值, 而 load 操作被阻塞住了。要想在 load 操作之后继续执行, 更严格的一个条件是要求该 load 操作必须全局完成, 这需要等到产生该 load 值的 store 操作已经全局完成 (Globally Performed, GP)。后面我们会用 GP 值来指代全局完成的 store 的值。

定义 7.4 (store 原子性的充分条件) 当同时满足如下条件时, 我们认为存储系统满足 store 原子性:

- 对所有地址的 store 操作可以确保形成一个全局序。
- 所有的 load 操作都必须全局完成。

这个充分条件给出了一个实现 store 原子性的可行方法。只要还有一个线程读到旧值, 那么其他线程也就不可以读到新值, 因此 store 操作看起来就好像正好在全局完成的那个时间点上原子执行的一样。注意这个充分条件没有规定处理器核具体以什么方式处理 store; 这意味着即使 store 还没有全局完成, 处理器核也还可以继续处理其他操作。这一特征非常关键, 因此图 7-3 中的 store 缓冲可以发挥作用。需要记住的是, 同一个线程内部对相同地址的 load 和 store 必须按照线程序执行, 以确保线程内部的依赖关系。在本章中, 始终假定这个条件是默认成立的。

图 7-15 展示了在一个满足 store 原子性的系统中, 数据值随时间的变化情况。两个不同线程的 load 和 store 操作的生命周期在图中用间隔表示。需注意的是, store 操作相关的行为在 store 全局完成之后仍然可能还继续存在, load 值锁定的时间点也各不相同。图中灰色的竖线表明了 store 操作全局完成所对应的时间点, 线程 1 的第一个 load 必须返回 a_1 , 这是因为在 load 的生命周期中返回的值是锁定不变的, 而其他两个 load 所返回的值则依赖于它们对值进行锁定的时间以及在锁定值的时刻哪个值已经全局完成。线程 2 的第一个 load 只能返回 a_0 或者 a_1 , 而线程 1 的第二个 load 则只能返回 a_1 或者 a_2 , 请记住 load 只能返回已经全局完成的值。

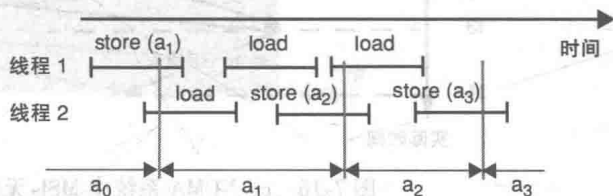


图 7-15 store 原子性系统中的全局完成值顺序

要想对相同存储地址的 store 操作进行全局定序, 在 SMP 系统中可以通过总线实现, 在 cc-NUMA 系统中则可以通过目录控制器 (通过目录中的 busy 位进行定序) 实现。而对于 load 操作来讲, 要进行全局定序似乎更加困难, 因为没有谁会通知处理器 load 的值已经全局完成了, 不过, 在一些重要和常见的实际情况中, load 全局完成的条件其实比较容易满足。前面我们已经介绍了在基于侦听总线、支持 MSI-无效协议的 SMP 系统中的一致性实现, 下面我们将根据定义 7.4 介绍在 cc-NUMA 和目录协议下的一致性实现。

例 7.2 cc-NUMA 中的 store 原子性 考虑图 7-11a 中的 cc-NUMA 协议事务, 在 MSI-无效协议中, 当存在两个远程 Shared 状态副本的时候, 会发生写失效。参考图 7-16, 线程 T0 (请求者) 在时间 t_0 发送一个 BusRdX 到主节点。主节点在 t_1 时接收到这个请求, 然后通过设置 busy 位来锁定该目录项。通过对内存块所对应目录项的锁定, 主节点确保没有任何线程可以重新获得这个内存块的副本, 因此确保了对这个块的所有 store 操作可以进行全局定序。然后, 在时间 t_1 , 主节点向 T0 的 cache 发送了一个块副本, 该副本在 t_2 时刻被接收到, 同时, 主节点还向 T1 和 T2 的 cache 发送无效信号, 到 t_3 时主节点接收到了 T1 和 T2 返回的所有无效应答, 于是在目录项中清空 busy 位, 然后发送信号给 T0, 告知其对应的 store 访问已经全局完成了, 该信号在 t_4 时刻被 T0 接收到。通过在 t_3 时刻对目录项进行解锁, 之后所有线程就都可以获取到 store 的新值了。T0 在 t_4 时完成了对应的 store 操作, 然后继续执行其他指令。

根据定义 7.2, 上述处理过程是满足 store 原子性的。在 cache 接收到无效信号之前, T1 和 T2 读到的一直是最新的 GP 值。当 cache 接收到无效信号并将对应数据块副本无效后, 会再发送应答消息给主节点。这时它们无法再访问到旧的 GP 值, 必须通过 cache 失效来获取到新值, 而 cache 失效只能在 busy 位被清空之后才能完成。

根据定义 7.2, 上述处理过程是满足 store 原子性的。在 cache 接收到无效信号之前, T1 和 T2 读到的一直是最新的 GP 值。当 cache 接收到无效信号并将对应数据块副本无效后, 会再发送应答消息给主节点。这时它们无法再访问到旧的 GP 值, 必须通过 cache 失效来获取到新值, 而 cache 失效只能在 busy 位被清空之后才能完成。

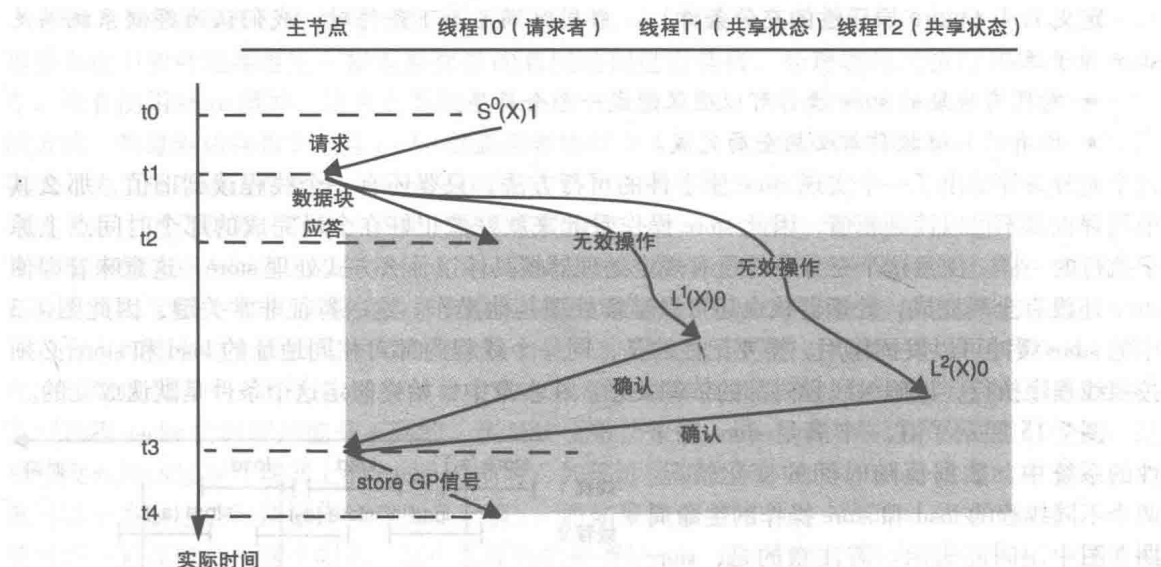


图 7-16 cc-NUMA 系统中 MSI-无效协议下的 store 失效处理

再考虑一个更加“放松”的策略。假设在 t_2 时接收到块副本， T_0 （以及其他共享其 cache 的线程）认为它获得了一个修改状态的副本，并且将 store 操作值写入 cache 中的 X 。然而，这个 store 还不是 GP 的，因为其他线程（比如 T_1 和 T_2 ）仍然可以读取到 X 旧的 GP 值。由于 cache 一致性是按照块级别来维护的，针对该块本地副本的后续其他 store 操作也都有类似的问题。

根据定义 7.4 中的充分条件，要想保证 store 的原子性，那么在 t_2 和 t_4 之间， T_0 上的任何对 X 的 load 操作都不允许返回 store 的最新值，因为不同线程可能会看到两个不同的值。如果这个数据块的其他位置在 t_2 之后也被修改了，那么上述情况同样适用。当运行线程 T_0 的处理器核同时也被其他线程共享时（比如核内多线程结构），或者当 cache 被多个核共享时，这一限制会更加严格。因为在这种情况下，cache（被多个本地线程共享）的块副本可以被所有的共享线程使用，并且这些共享线程都可以在这个数据块上进行 load 或 store 操作，因此导致多个线程对于非 GP 值的扩散使用。

在更新协议下，store 原子性的充分条件更加难以保证。图 7-11b 显示了一个共享状态数据块上发生 store 命中时的一致性处理过程，该共享状态数据块有两个远程共享副本。因为更新操作传播到 T_1 和 T_2 可能有不同的延迟，在很长的一段时间里， T_1 可能读取到新值，而 T_2 还在读旧值。这暴露了硬件上缺乏对 store 原子性的保证。通常的解决办法是，让主节点的目录控制器给 T_1 和 T_2 发送更新值，并将更新值锁定在它们的 cache 中。这时没有哪个线程能再返回“旧”数据，因此 store 操作全局完成。等 T_1 和 T_2 的更新应答返回到主节点之后，主节点会通知 T_0 其 store 操作完成，同时发送第二波消息给 T_1 和 T_2 的 cache，将新值进行解锁。在 t_0 和 t_4 之间， T_0 的 cache 更新一直被锁定，图 7-17 显示了整个操作的原子事务过程。

在图 7-16 和图 7-17 中的协议处理事务中，当 load 操作的值从存储系统返回并被锁定时，该操作就已经自动全局完成。这类存储系统保证了访存操作的原子性，我们称之为原子存储系统。

7.3.4 纯一致性

store 操作原子性的条件非常严格，在实际中可能并不一定是必需的。一致性问题的出现是

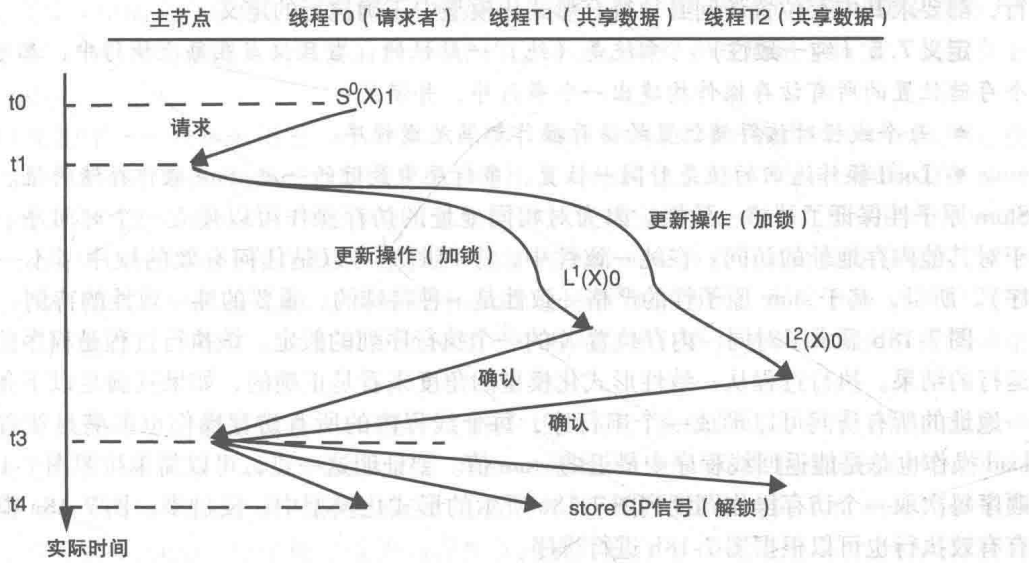


图 7-17 cc-NUMA 中 MSI-更新协议下的原子 store 事务

由于系统中对应同一位置的数据可能在多个地方存在多个副本，要证明系统中的多个副本是满足一致性的，只需要证明该系统和只存在单一副本的系统是等价的即可。我们将这种放松形式的一致性叫作纯一致性，或者简称为一致性，以区分之前介绍的严格一致性以及 store 原子性。

一致性的形式模型

图 7-18a 显示的是传统的一致性形式模型，在该模型中，多个线程共享单个内存位置 X。除了访问 X 的操作，线程还执行其他指令，比如算术操作、浮点运算、分支/跳转，以及对其他地址的访问等，但这些指令都不属于形式模型的一部分。

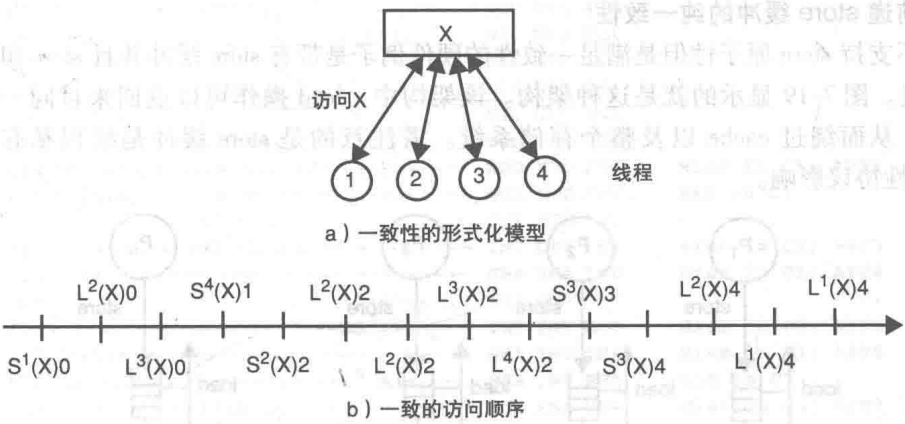


图 7-18 一致性的形式化模型和一致的访问顺序

形式模型的规则如下：

- 在同一时刻，每个线程只能有一个对内存位置 X 的访问，且多次访问之间按线程序进行。
- 在同一时刻，只能有一个针对内存位置 X 的访问在执行并完成。

由定义可以看出，系统对每个内存位置进行访问时，如果都能按照线程序正确执行，就像在只有单一副本的系统上执行时一样，那么该系统就是一致的。在目标系统上的每一次正确执

行，都要求其内存位置访问必须符合形式化模型中正确执行的定义。

定义 7.5 (纯一致性) 系统是 (纯) 一致性的，当且仅当在每次执行中，都可以针对每个存储位置的所有访存操作构建出一个串行序，并满足：

- 每个线程对该存储位置的访存操作都满足线程序；
- Load 操作返回的值是对同一位置、串行序中最近的一次 store 操作存储的值。

Store 原子性保证了严格一致性，因为对相同地址的访存操作可以建立一个时间序。而不依赖于对其他内存地址的访问。在纯一致性中，序列顺序可以是任何有效的顺序 (不一定是时间序)。所以，基于 store 原子性的严格一致性是一种特殊的、重要的纯一致性的特例。

图 7-18b 显示了对同一内存位置 X 的一个执行序列的假定，该执行过程是程序的一次特定运行的结果。执行过程从一致性形式化模型的角度来看是正确的，如果其满足以下条件：对同一地址的所有访问可以形成一个串行序，每个线程内的所有访存操作也都满足线程序，并且 load 操作也总是能返回线程序中最近的 store 值。要证明这一点，可以简单按照图 7-18b 所示的顺序每次取一个访存操作调度到图 7-18a 所示的形式化模型中。反过来，图 7-18a 模型中的所有有效执行也可以根据图 7-18b 进行排序。

要证明系统满足一致性，那么对于所有可能的执行过程，需要针对相同位置的所有访存操作找到一个对应的串行序，而这已经被证明是一个 NP 完全问题，因此实际难以实现。更重要的是，程序的执行过程是不可计算、难以判断的 (类似测试)，因为可能的执行情况数量是无穷的。因此，实际硬件中必须限制可能的执行情况，并且确保对于所有可能的执行情况都能建立一个串行序。要实现这一点，通常需要在访存硬件中增加一些串行点或者瓶颈点 (比如总线或者目录锁)。

下面我们探讨一种不支持 store 原子性，但是仍然满足 (纯) 一致性的存储系统，在这一部分中，将介绍如何基于硬件上的限制，为任意位置的访存执行都构建一个串行序，进而通过这种系统性方法来检测系统的一致性。

支持前递 store 缓冲的纯一致性

一个不支持 store 原子性但是满足一致性的硬件例子是带有 store 缓冲并且 store 可以前递给 load 的系统。图 7-19 显示的就是这种架构。该架构中，load 操作可以返回来自同一线程 store 缓冲的值，从而绕过 cache 以及整个存储系统。需注意的是 store 缓冲是线程私有的，不受 cache 一致性协议影响。

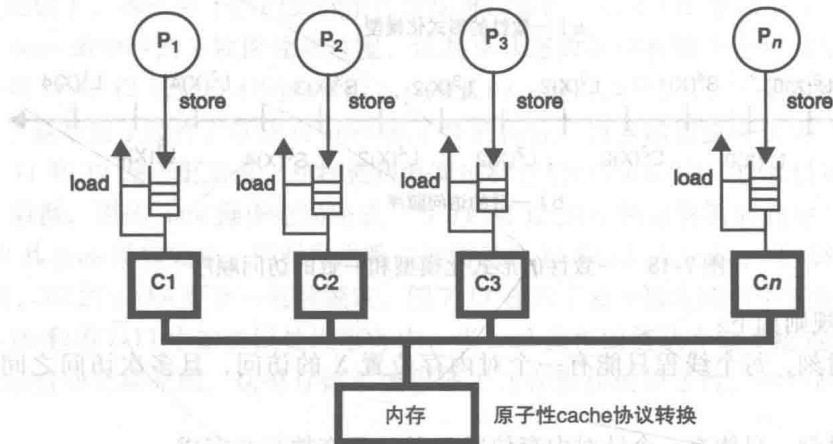


图 7-19 支持前递 store 缓冲的多处理器

为了简化问题，重点考虑支持前递的 store 缓冲的影响。假定所有的 cache 访问都是原子性

的，根据定义 7.4 中的充分条件，整个系统不满足 store 原子性，因为 load 操作可能返回非 GP 的值，并且不同线程可能在同一时间读取到不同的值。不过，系统仍然满足一致性，我们在下面将会进行说明。

图 7-19 架构中的处理器采用图 7-3 中所示结构，store 缓冲在写回数据的 cache 之前，在 MSI-无效协议下，store 操作在 store 缓冲中完成并退出。对相同地址的多个 store 操作会在 store 缓冲中进行合并，如果 store buffer 中已经有了某个内存地址对应的项，那么之前的旧的 store 会被覆盖掉，以保证 store 缓冲中保存的是对应每个地址的最新 store 操作。load 操作在访问 cache 时优先级比 store 更高，如果在 store 缓冲中没有 load 地址所对应的项，那么 load 也可以跳过 store 缓冲中待处理的 store 操作；而如果在 store 缓冲中有对应项，那么 store 的值会直接前递给 load 操作。

图 7-20 中给出了一个支持前递 store 缓冲（图 7-19 所示）系统执行的例子。除了线程访问，我们也给出了 cache 原子访问的时间点。当 store 操作从 store 缓冲中移出的时候，它对 cache 进行更新。我们用 WB 来表示这些 cache 更新。大部分访存操作（load 和 store）都是在 store 缓冲中本地完成，cache 中全局完成的访存操作用粗体标示了出来。一部分 cache 访问命中，而其他的 cache 访问则引发了原子协议事务（Atomic Protocol Transaction, APT）。这个执行示例表明，在支持前递 store 缓冲的系统中，store 操作不是原子性的，因为在大多数情况下，不同处理器的 load 操作可以同时返回不同的值。如果我们不清楚产生这一现象的硬件结构特征，那么很可能会认为存储系统完全是无序的。

T1	T2	T3	cache 状态			备注
			C1	C2	C3	
t0-----		L ³ (A) a ₀ ----	NIC	NIC	SHA	Miss in C3; APT1
t1 S ¹ (A) a ₁			NIC	NIC	SHA	
t2		S ³ (A) a ₂	NIC	NIC	SHA	
t3 L ¹ (A) a ₁			NIC	NIC	SHA	
t4	S ² (A) a ₃		NIC	NIC	SHA	
t5		L ³ (A) a ₂	NIC	NIC	SHA	
t6 S ¹ (A) a ₄			NIC	NIC	SHA	
t7	L ² (A) a ₃		NIC	NIC	SHA	
t8 L ¹ (A) a ₄			NIC	NIC	SHA	
t9	L ² (A) a ₃		NIC	NIC	SHA	
t10 WB ¹ (A) a ₄ -----			MOD	NIC	INV	Miss in C1; APT2
t11 L ¹ (A) a ₄ -----			MOD	NIC	INV	Hit in C1
t12	S ² (A) a ₅		DTY	NIC	INV	
t13-----WB ² (A) a ₅ -----			INV	DTY	INV	Miss in C2; APT3
t14 L ¹ (A) a ₅ -----			SHA	SHA	INV	Miss in C1; APT4
t15		L ³ (A) a ₂	SHA	SHA	INV	
t16-----WB ³ (A) a ₂ ----			INV	INV	MOD	Miss in C3; APT5
t17 L ¹ (A) a ₂ -----			SHA	INV	SHA	Miss in C1; APT6
t18-----L ³ (A) a ₂ ----			SHA	INV	SHA	Hit in C3
t19-----L ² (A) a ₂ -----			SHA	SHA	SHA	Miss in C2; APT7
t20	S ² (A) a ₆		SHA	SHA	SHA	
t21	L ² (A) a ₆		SHA	SHA	SHA	
t22-----WB ² (A) a ₆ -----			INV	MOD	INV	Upgrade in C2; APT8

图 7-20 支持前递 store 缓冲系统中的执行

即使在大多数情况下，不同线程的 load 操作会返回不同的值，但该系统仍然满足一致性，因为针对某一特定内存位置的所有执行都可以按照图 7-18 中的形式化模型来表示成有效执行。要证明这点，需要有一个系统性方法——可以为同一地址的所有访存生成一个全局序。原子的 cache 系统刚好提供了一个硬件上的“瓶颈”点，可以作为实现全局序的关键。

首先,我们从对所有全局完成 (GP) 的访问进行定序开始,比如 cache 中的原子执行操作:

$$\begin{aligned} L^3(A)a_0 < WB^1(A)a_4 < L^1(A)a_4 < WB^2(A)a_5 < L^1(A)a_5 < WB^3(A)a_2 \\ < L^1(A)a_2 < L^3(A)a_2 < L^2(A)a_2 < WB^2(A)a_6 \end{aligned}$$

上述顺序中, WB 不是线程中的访问,而是标记了从 store 缓冲对 cache 进行更新的时间点。对 cache 的更新操作是随机发生的 (只要 cache 带宽足够),其时序取决于对其他内存位置的 store 操作。

第二步是将全局序中的 WB 进一步扩展为正在退出的 store 缓冲项中的所有 load 和 store:

$$\begin{aligned} L^3(A)a_0 < S^1(A)a_1 < L^1(A)a_1 < S^1(A)a_4 < L^1(A)a_4 < L^1(A)a_4 < S^2(A)a_3 \\ < L^2(A)a_3 < L^2(A)a_3 < S^2(A)a_5 < L^1(A)a_5 < S^3(A)a_2 < L^3(A)a_2 \\ < L^3(A)a_2 < L^1(A)a_2 < L^3(A)a_2 < L^2(A)a_2 < S^2(A)a_6 < L^2(A)a_6 \end{aligned}$$

在这个全局序中,来自每个线程的访问符合线程序,load 操作按顺序返回最新的 store 的值,因此图 7-20 的执行过程是满足一致性的。需注意的是一致性序和值的时间序是不一样的。

- 值的时间序为: $a_0 < a_1 < a_2 < a_3 < a_4 < a_5 < a_6$

- 值的一致性序为 $a_0 < a_1 < a_4 < a_3 < a_5 < a_2 < a_6$

不过所有线程观察到的值的顺序是相同的,都符合一致性序:

- T1 观察 $a_1 < a_4 < a_5 < a_2$

- T2 观察 $a_3 < a_5 < a_2 < a_6$

- T3 观察 $a_0 < a_2$

线程在一致性序中会跳过一些值,这是因为它们没有对应的 load 操作可以观察到这些值。不过,它们能观察到的值都是满足一致性序的。我们的结论是:所有支持前递 store 缓冲的系统都是满足一致性的。

一般化推广

上面基于前递 store 缓冲分析所获得的经验可以做直接的推广,这里不再给出繁琐的例子,形式化验证会作为一个习题放在后面。这里可以先定义一个关于一致性的“私有原则”:在某个最新的 store 值传播到其他线程之前,线程可以通过 load 获取并使用该最新的 store 值,而不会违反一致性。原因在于其他线程观察不到这个本地值,我们可以在这个本地值变成 GP 时,直接将这个本地访问插入到全局序当中。

在例 7.3 中,我们在 store 缓冲中使用了非常激进的策略将所有相同地址的 store 进行了合并,我们也可以不使用这么激进的策略,只要 load 可以获取到最新的 store 值,那么这种 store 缓冲仍然是可以满足一致性的。当然,如果 store 缓冲不支持前递,也不会违反 store 原子性,这种情况下,load 必须总是在 cache 中获取到 GP 值后才能完成。

我们还可以按照下列方式做进一步的一般化推广,在逻辑上和支持前递的 store 缓冲结构是等价的,并且也支持纯一致性,但是不支持 store 原子性。

- 无锁 cache 中,在 store 失效还未完成的时候,可以分配新的 cache 行,本地的 load 和 store 操作也可以读写该 cache 行。

- 多线程 CMP 系统中,共享同一个核的多个线程,或者共享同一个 cache 的多个处理器核,都可以在 cache 中互相读取到对方的值,即使这个值还没有完成 (不是 GP 的)。直觉上,我们也可以通过在各层次 cache 中应用上面介绍的“私有原则”获得上述结论。

- 在多层 cache 系统中,核组或者多核之间可以在各层次的共享缓冲区或者共享无锁

cache 中对一些非 GP 值进行共享。

- 在支持无效协议的 cc-NUMA 系统中，即使目录中对应的一致性事务还没有完成，线程也可以对本地接收到的数据块进行修改并使用修改值，也可以在其他本地线程之间共享这些值。如图 7-16 中的例子所示，图中的灰色区域描述了这类系统中纯一致性和 store 原子性之间的区别。

通常情况下，线程或者线程组在存储系统中有一个 store 流水线，流水线中保存了来自线程或者线程组的所有未完成的 store 操作，即使对应的 store 值还没有完成全局广播，线程或线程组也可以从 store 流水线中读取数据，这不会违反纯一致性。

和定义 7.4 中的 store 原子性相比，这里需要针对所有 store 操作找到一个可串行化的点，这是一致性存储系统中的重要部分。这个串行化点是实现所有可能排序的关键。store 操作的串行化可以在不同级别上进行，比如在 CMP 系统上，或者在集群系统/多芯片多处理器系统级别上。不管在什么级别上，存储操作最终都必须全局排序。假如 load 的值最终也可以全局排序，那么 load 操作就不需要等到对应的 GP 值才能完成。因此，在一致性存储系统中，load 操作可能比在 store 原子性存储系统中更早提交。在一致性存储系统中，load 操作只需要将它的返回值锁定到最近的一致性副本中即可，即使这个值还不是 GP 的也没问题。而在 store 原子性系统中，load 操作返回的值必须是 GP 的。

纯一致性的重要性

乍看起来，（纯）一致性的条件似乎非常弱，因此，人们可能会问：是否存在某种执行情况不满足一致性？

为了阐述一致性的主要影响，我们先回顾一下纯一致性的定义：对于所有的执行情况，必须为同一地址的所有访问找到一个一致的串行序，以确保线程内的所有访问满足线程序。下面使用三个例子来说明，所有例子中都访问相同的内存位置。

例 7.3 线程 T2 观察到线程 T1 两个 store 操作的乱序执行结果如下，证明这个执行过程不满足一致性。

T1	T2
S ¹ (A) a ₁	L ² (A) a ₂
S ¹ (A) a ₂	L ² (A) a ₁

在上面的例子中，T2 的 load a2 操作必须在 T1 中的 store a2 操作之后。并且由于在任何有效的一致性序中，T1 的两个 store 操作都必须满足线程序，因此 T2 的第二个 load 操作是不可能返回 a₁ 的。

例 7.4 线程 T1 和 T2 的 load 操作分别获取到对方的 store 操作结果，证明这个执行过程不满足一致性。

T1	T2
S ¹ (A) a ₁	S ² (A) a ₂
L ¹ (A) a ₂	L ² (A) a ₁

如果 T1 的 load 操作返回的是 a₂，那么在一致性序中，T2 的 store 操作一定是在 T1 的 store 和 load 操作之间执行，在这种情况下，T2 的 load 操作不可能返回 a₁，因为 a₁ 对应的 store 操作早于 a₂ 对应的 store 操作。

例 7.5 线程 T3 和 T4 观察到线程 T1 和 T2 两个 store 操作具有不同的顺序，证明这个执行过程不满足一致性。

T1	T2	T3	T4
$S^1(A) a_1$	$S^2(A) a_2$	$L^3(A) a_1$	$L^4(A) a_2$
		$L^3(A) a_2$	$L^4(A) a_1$

T3 的 load 操作先返回 T1 store 的值, 然后再返回 T2 store 的值, 而 T4 的 load 操作观察到的两个 store 操作的值的顺序刚好相反。我们无法为上述 6 个访存操作构造出一个满足一致性的顺序。

图 7-18 所示的一致性实现中包含了存储系统和处理器, 通过下面这个例子, 我们可以看到一致性的实现是需要非常谨慎的。

例 7.6 支持推测乱序执行的处理器违反纯一致性 除非采用特别处理, 否则乱序执行 (OoO) 处理器中的 load/store 队列可能会违反纯一致性。再次考虑例 7.3 中的代码, T2 的 load 观察到 T1 的 store 操作乱序完成, 注意 T2 这两个 load 的地址相同。这种现象在支持推测乱序执行处理器的 load/store 队列中是可能发生的, 因为 T2 中两个 load 操作的地址可能来自于不同的地址寄存器。在这个例子中, T2 的第一个 load 操作可能由于等待地址而在 load/store 队列中被延迟处理, 而第二个 (最新的那个) load 操作却已经就绪。除非在 load/store 队列中采取特殊处理, 否则第二个 load 操作可能返回 T1 的第一个 store 操作的值, 然后, T2 的第一个 (最老的) load 操作返回 T2 的第二个 store 的值, 正如例 7.3 中描述的那样, 这种现象违反了纯一致性。为了避免这个问题, 远程的 store 操作必须侦听 load 队列, 并且对具有相同地址的项进行标记。load 操作在发送到 cache 之前, 必须检查是否存在更新的 load 操作, 如果发现有更新的带标记的 load 项已经返回了值, 那么执行过程必须进行回滚, 因为继续执行会违反纯一致性。在第 3 章中, 我们没有对 load 发送到 cache 后的上述额外操作进行介绍, 这是因为 load-load 依赖关系并不会在单处理器中引起任何冲突, 不过, 在多处理器环境中, 这种机制就是必需的。

上述例子简单说明了纯一致性的形式化定义, 通过这一定义, 系统必须强制为相同地址的所有 store 操作建立一个全局序。除了确保这一特性之外, 纯一致性的价值还体现在以下几个方面:

- 对纯一致性的支持有助于存储一致性的实现。借助纯一致性的支持可以及时、有选择地、高效地 (借助一致性协议提供的过滤机制, 以及快速前递请求和应答的硬件机制) 完成值的传播, 存储一致性的内容将在本章后面部分进行介绍。
- 对纯一致性的支持可以确保当计算过程突然中止时 (比如碰到上下文切换), 在所有处理指令执行完毕, 并且网络和所有的缓冲区都已经排空之后, 存储系统可以进入到一个一致的状态。在图 7-20 的例子中, 新的上下文开始运行之前, 所有 store 缓冲中的值都必须全局完成, 因此在上下文切换之前, 需要在整个系统中全局完成所有的值并进行排序 (使用和图 7-20 中建立全局序完全一样的步骤)。
- 和上面的原因类似, 纯一致性还能解决由于线程迁移所引起的存储相关问题。

纯一致性的问题

纯一致性不支持对不同地址的访存操作顺序进行组合, 换句话说, 假定有两个变量的访问, 并且每个变量的访问过程单独看起来都是满足一致性的, 但是, 在两个变量访问之间所引入的其他任何顺序性都可能会导致整个执行过程不可定序。而要证明一个执行过程不可定序比证明其可定序要复杂得多, 比如, 大多数的形式化验证方法也都假定所有的访存操作存在一个一致性序。

例 7.7 纯一致性序和其他序的冲突 考虑以下包含两个地址 A 和 B 的执行过程:

INIT A=0; B=0;

T1	T2
$S^1(A) 1$	$S^2(B) 1$
$L^1(A) 1$	$L^2(B) 1$
$L^1(B) 0$	$L^2(A) 0$

根据纯一致性的形式模型，上述执行对于 A 或者 B 的访问单独来看都满足纯一致性，因此是正确的。然而，当存在下列情况时，纯一致性序会和每个线程强加在这两个 load 上的其他序产生冲突。

- 情况 1：存储模型中可能会存在一个常见的限制，那就是在相同线程中的两个 load 操作上强制建立一个全局序，即使它们的访存地址不相同。
- 情况 2：第二个 load 操作依赖于第一个 load 操作，比如第二个 load 操作的地址计算可能需要第一个 load 返回的值。此时在第一个 load 操作返回值之前（线程内的数据依赖），第二个 load 操作甚至都不能开始执行或者是预取。
- 情况 3：程序员可能在两个线程中的两个 load 操作之间插入一个 barrier 同步，因此在每个线程的两个 load 操作之间都强制添加了一个时间序。

在上面的三种情况中，线程中的两个 load 操作会被强制串行顺序执行，然而这个顺序会和一致性序产生冲突，导致整个执行过程无法进行全局定序。在上面的代码中，满足一致性的机器会针对所有地址的访问在线程中维护一个线程序。在 T1 中，load A 操作必须在 store A 操作之后（由于一致性），由于其他顺序性的约束，load B 操作必须在 load A 操作之后。由于 A 的一致性序要求，T2 中 load A 操作必须早于 T1 中 store A 操作。因此当所有的顺序性都满足的时候，T1 中 load B 操作是不可能返回 0 的。

在支持 store 原子性的系统中，上面例子中的执行过程也不可能出现，因为 store 原子性能够实时地对相同地址的访问进行定序，在 T1 执行 A 和 B 的 load 操作之前，T1 的 store A 操作的结果必须已经传播到所有的副本中了（包括 T2 中的 cache）。◀

例 7.8 同时满足纯一致性和 store 原子性的执行 下面的执行过程是否满足纯一致性？是否满足 store 原子性？

INIT A=0; B=0;

T1	T2
S ¹ (A) 1	S ² (B) 1
L ¹ (B) 0	L ² (A) 0

首先，这个执行过程肯定是满足纯一致性的，因为对每个地址只有两次访问操作，并且是在不同的线程。此外，它也满足 store 原子性。通过定义 7.4 中 store 原子性的充分条件可知，即使 store 操作没有全局完成，后续执行也可以继续推进，但是 load 操作必须等到它对应的值全局完成后才能继续。这个例子中，线程对两个不同地址的 store 操作可以在存储系统中同时并发传播，相互交错，在对方线程的 load 操作全局完成之后，再到达其 cache 中。（注意这两个 store 操作是针对不同地址的。）

即使在两个线程的 store 和 load 操作之间加入一个 barrier（要想获得确定的结果，这是一个必要的步骤），结果仍然是满足纯一致性和 store 原子性的，但是这个结果是不正确的：

INIT A=0; B=0;

T1	T2
S ¹ (A) 1	S ² (B) 1
BARRIER(bar1)	BARRIER(bar1)
L ¹ (B) 0	L ² (A) 0

store 原子性这一属性所针对的是单个内存地址，它对其他地址的访问没有约定，哪怕是 barrier 操作。因此，除了 store 原子性，还需要其他的一些机制来避免出现上述的错误执行结果。换句话说，硬件必须能够识别出此类 barrier 操作，并且提供一种机制，确保在 barrier 执行之前，所有的 store 操作都已经全局完成了（如图 7-16 和图 7-17 中发送给线程 T1 的 store 操作 GP 信号）。◀

7.3.5 store 原子性和访存交错

纯一致性的形式化模型所引入的执行顺序有时候和线程间访存依赖所引入的其他实时限制会产生冲突，这一点我们在上面的例 7.7 中已经看到了。

图 7-21 所示的模型考虑了所有地址的访问操作，每一个地址分配了一个独立的内存。线程可以按任何顺序发出内存访问，只要不违反线程内的依赖关系，并且和访问其他地址时可能引入的顺序性要求一致即可。内存逐个响应每个访存请求（load 或者 store），因此每个地址的访问都是原子性的，但是每个线程可能同时有多个针对不同地址的未完成访存请求，而内存也可以被所有线程并行访问。

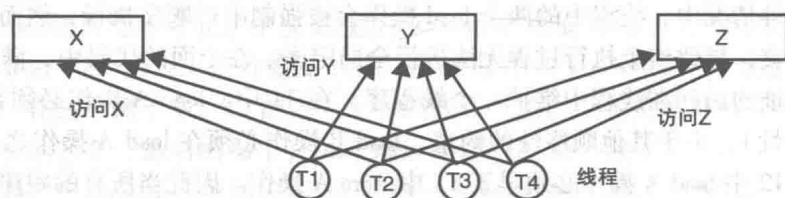


图 7-21 store 原子性和访存交错模型

定义 7.2 ~ 定义 7.4 中的条件形成了一个可以用图 7-21 进行建模的系统。在上述条件中，要求对相同地址的 store 操作顺序进行处理，在任何时刻，load 操作只能观察到并返回一个值。这和图 7-21 所示模型中的内存系统是完全相同的：load 操作实时观察到一系列的 store 值，并且同一时刻只能访问到一个值，具体的访问顺序和访存瓶颈处所决定的时间序一致。

图 7-21 中的模型实际上是一个交错访存的系统，系统中所有内存地址都只有单一副本。在这种存储系统中，对所有地址的所有 store 操作可以构成一个全局序，关于这点可以简单说明如下：

- 如果两个访存操作是对同一地址的，或者地址不同但是两个访存不是并发的（即在时间上没有重叠），那么这两个操作就可以按照访存的真实时间进行排序，同时也能保证正确的执行结果。
- 如果对不同地址的两个访存操作在分别访问各自内存时存在并发，由于操作已经在内存中了，它们不会相互影响到对方的地址或者输入值，所以此时访存的所有输入操作数必须已经准备好。因此对不同地址的两个并发访存操作可以任意排序（比如，可以根据开始时间排序），其执行结果仍然是正确的。

这种特性并不奇怪，众所周知，交错存储系统等效于一个单体、集中式的存储器。和纯一致性相反，store 原子性是可组合的，换句话说，我们可以简单地对每个单独内存地址构造一个 store 顺序，进而为所有内存地址的所有 store 操作构造出一个全局序。因此，对所有地址的所有访存构造一个全局序是可行的，并且是可扩展的，因为 store 原子性并不需要针对所有 store 操作创建一个串行瓶颈点，而只需要单独考虑每个内存地址的 store。这是一个非常关键的特征。

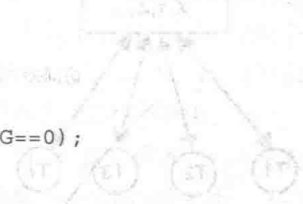
7.4 顺序一致性

纯一致性和 store 原子性都是存储系统中非常有用的特征，但是它们还不足以保证执行过程的正确性，因为针对不同共享地址的访问可能也会相互影响，而纯一致性以及 store 原子性都只考虑对相同地址访存操作的顺序。

为了进一步说明，下面给出一些利用了图 7-1 中共享内存通信模型的代码。

• 点对点通信

```
INIT: A=FLAG=0
T1      T2
A=1 ;   while (FLAG==0) ;
FLAG=1; Print A;
```



在这段代码中，程序员期望 T1 对 A 的更新在 FLAG 更新之前能到达 T2，这样 T2 打印出的 A 的值总是 1。

• 值通信

```
INIT: A=B=0
T1      T2
A=1     Print B;
B=1     Print A;
```

在这段代码中，如果 B 的打印结果是 1，那么 A 也同样应该打印出 1。

• Dekker 算法

```
INIT A=B=0
T1      T2
...     ...
A=1     B=1
while (B==1);   while (A==1);
<critical section> <critical section>
A=0         B=0
```

在这段代码中，线程可能互斥地执行临界区内的代码，也可能是两个线程同时执行 while 语句从而出现死锁，这两种情况都是程序员所允许的，但是程序员不希望看到两个线程同时执行临界区内的代码。刚刚在例 7.8 中，我们看到线程 T1 和 T2 中对 A 和 B 地址的 load 操作都可能返回 0（即使存储系统满足 store 原子性），在这种情况下，两个线程会同时进入临界区。

在上面这些代码片段中，程序员直觉上认为，不同线程的多个访存指令的效果，就像是按照线程程序在真实时间上交错的原子执行一样，这种模型我们称之为顺序一致性。从软件的角度来看，这是一个非常严格的要求。正如名字所表明的那样，任意多线程程序的执行过程必须跟所有指令在单个线程中进行串行交错执行的效果一致。不管针对哪种多线程硬件环境，这一条件都是非常苛刻的。顺序一致性是计算机系统最严格的存储一致性模型。之所以说是最严格的模型，是因为硬件已经无法为程序员提供更多的支持了。顺序一致性模型也是在硬件上限制最多的一个存储模型，由于这个原因，我们也经常称之为“强一致性”或“强顺序性”。

7.4.1 顺序一致性的形式化模型

顺序一致性是一种描述并发硬件行为的模型，来自不同线程的访存操作以线程程序的方式原子地交错执行。图 7-22a 给出了顺序一致性的形式化模型。

每个线程执行的指令包括共享内存访问指令以及其他指令，而在形式化模型中我们只考虑共享内存访问指令，形式化模型的规则可以总结如下：

- 每个线程以线程程序逐一执行共享内存访问操作。
- 同一时刻只能有一个对共享内存访问的操作在执行。

根据上述定义，如果系统中的每次执行都符合图 7-22a 中形式化模型的合法执行，那么这个系统是顺序一致性的。

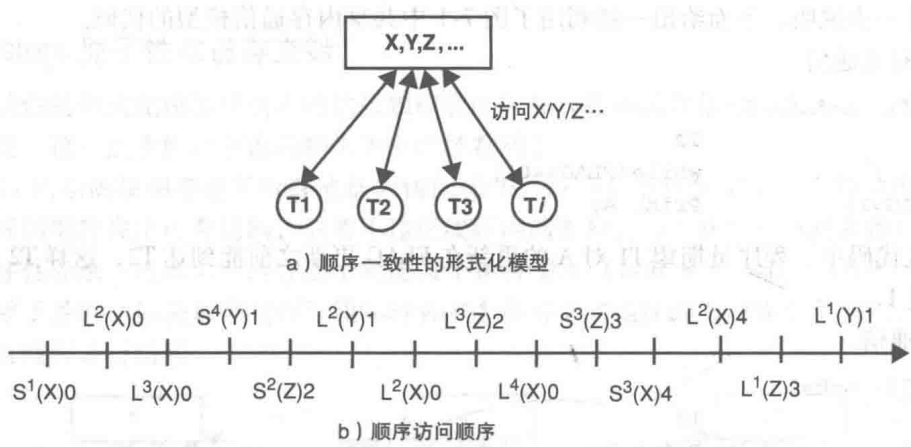


图 7-22 顺序一致性的形式化模型和顺序访问顺序

定义 7.6 (顺序一致性) 多处理器满足顺序一致性，如果任意一次执行的结果都好像是所有线程的访存操作按照某种串行的顺序执行，并且每个线程内的执行过程都符合线程序。

在这个定义中，“任意一次执行”指的是“任意程序的任意一次执行”，“结果”指的是所有 load 操作返回的值。因此，对于每次执行过程，要证明系统满足顺序一致性，需要针对所有线程对所有地址的所有访问操作构建出一个一致的串行序。这个过程和纯一致性的判定类似，区别在于这里需要为所有内存地址的访问都构建一个全局序。

图 7-22b 给出了一个假想程序执行过程的访存顺序，其中每个线程的访存都符合线程序，并且 load 操作总是返回全局序中最近 store 的值。因此这个执行在形式化模型看来是一次合法的执行。要证明这一点，可以简单地按顺序逐个选中每个内存访问，然后将其按照形式化模型进行调度。由于软件通常是感知不到实际时间的，因此软件检测不到它是在一个满足顺序一致性的多线程系统上执行还是在一个形式化模型上执行。

例 7.9 违反顺序一致性的情况 定义 7.6 为很多理论上的优化方法提供了可行性，我们再以下面这段代码为例：

```
INIT: A=B=0
T1      T2
A=1     Print B;
B=1     Print A;
```

如果在真实时间上，T1 中对 B 的 store 操作在对 A 的 store 操作之前全局完成，那么这个执行还是顺序一致性的吗？

在顺序一致性约束下，A 和 B 合法输出的打印结果只能是 (A, B) = (0, 0), (1, 0) 和 (1, 1)，而 (A, B) = (0, 1) 这个输出是不可能出现的，因为如果 T2 没有观察到 A = 1，那么它也不可能观察到 B = 1。

再考虑下面这样一种实际的执行情况。由于某些原因，真实时间里 T1 可能在 A = 1 之前就先全局完成了 B = 1。如果 T1 希望尽可能快地传播 store 值，那么即使系统符合 store 原子性，上面这种情况也是可能出现的。比如，store A 在 cache 中失效了，而 store B 命中，那么 T1 的 cache 可能会先执行 store B，然后再执行 store A，这样 B 的新值可能会比 A 的新值先到达 T2。

乍看起来，上面这个执行过程不可能符合顺序一致性。然而，如果假定 T2 执行得比较慢，这两个打印语句也在 T1 对 A 和 B 的更新都已经全局完成之后才执行，因此 T2 的 load 操作和 T1 的 store 操作之间在真实时间上没有重叠，此时的输出是 (A, B) = (1, 1)，即使 T1 的两

个 store 操作是乱序全局完成的也是这样。这意味着这个执行过程仍然是顺序一致性的，因为它产生了一个满足顺序一致性的结果。类似地，如果 T2 的两个 load 乱序执行，并且都在 T1 的两个 store 操作之后执行，那么程序输出 $(A, B) = (0, 0)$ 这样的结果也是符合顺序一致性的。

不幸的是，在真实系统中，我们很难利用这种优化，因为 T1 必须先知道 T2 要到很久以后才会执行打印语句（反之亦然），这样 T1 才能乱序地全局完成这两个 store 操作。在真实系统中，我们不能假定线程之间一定存在这样一种可靠的关系，在无法保证这一点的情况下，上述优化就是不安全的。

和之前介绍的纯一致性的情况类似，我们也需要一个考虑了硬件结构特征的系统性方法，来为每次执行过程中的访存操作构造一个全局的一致序，和之前的区别在于，这里的全局序需要包括所有地址的访存操作。下面给出的是图 7-3 所示的五级流水线结构满足顺序一致性的充分条件。

定义 7.7（顺序一致性的充分条件）

- 对相同地址的 store 操作存在一个全局序。
- 只有当之前的所有访存操作（load 和 store）都已经全局完成时，线程才可以发射新的访存操作。

这组条件自动保证了 store 原子性，但是远比 store 原子性更严格。第二个条件强制处理器在执行任何地址的访存之前必须等待前面的访存操作完成，而 store 原子性的充分条件只要求 load 操作需等待相同地址的 store 操作完成，store 操作无需等待。不过，上述这组条件更可行，也更可扩展，因为它依赖的是针对每个地址和每个处理器的独立机制。

由于访存是以线程程序发射到内存中的，并且同一时刻只有一个访存全局完成，因此满足定义 7.7 的任意执行过程都是图 7-22 所示形式化模型中的有效执行。在图 7-3 的流水线中，这一条件意味着 store 操作可以插入到 store 缓冲中等待 cache 访问，而 load 操作必须等待直到 store 缓冲中之前所有的 store 操作都已经全局完成才能继续执行。load 也必须全局完成才能返回值，store 则必须按照先进先出的顺序从 store 缓冲中逐个全局完成。毋庸置疑，这一限制使得 store 缓冲几乎没有什么实际效果，因为实际程序中在 load 和 store 之间通常只有很少的其他指令。

7.4.2 顺序一致性的访存顺序规则

通常情况下，我们很难准确判断某次执行是否满足顺序一致性，但是借助一些可以在执行流图上进行验证的简单规则，计算机可以做到这一点。顺序一致性（s.c.）要求首先保证线程程序（t.o.），此外，load 相关的 store 操作必须先于 load 操作完成，我们概括为如下两个规则：

$$Op^i(A) \xrightarrow{t.o.} Op^i(B) \Rightarrow Op^i(A) \xrightarrow{s.c.} Op^i(B)$$

$$Val[S^i(A)]: Val[L^j(A)] \Rightarrow S^i(A) \xrightarrow{s.c.} L^j(A)$$

Op 指的是 load 或者 store 操作， \rightarrow 代表“先于”，Val 是 Op 的值，“:”表示 store 操作的值是 load 操作的源。

此外，还需要用到两个针对相同地址 load 和 store 操作的隐含规则，以确保顺序一致性：

$$(Val[S^i(A)]: Val[L^j(A)]) \text{.and.} (S^k(A) \xrightarrow{s.c.} L^j(A)) \Rightarrow S^k(A) \xrightarrow{s.c.} S^i(A)$$

$$(Val[S^i(A)]: Val[L^j(A)]) \text{.and.} (S^i(A) \xrightarrow{s.c.} S^k(A)) \Rightarrow L^j(A) \xrightarrow{s.c.} S^k(A)$$

这两个隐含规则表明，如果 store 操作和后续 load 操作有相关，那么在这个 store 和 load 操作之间，不允许插入其他针对同一地址的 store 操作。

基于测试程序可以创建一个执行流图，通过有向边来表示相应的规则，如果最后图中存在

环，那么该执行过程就无法按照顺序一致性进行定序。

例 7.10 证明下述执行过程不满足顺序一致性 顺序一致性无法真正利用 store 缓冲的优势，因为在 store 缓冲中，load 操作允许先于之前的 store 操作执行，这使我们无法为所有访存操作构建一个全局的一致序，从而无法满足形式化模型的要求。为了说明这一点，考虑如下的执行过程（这是 Dekker 算法的一部分），证明了其不满足顺序一致性：

INIT: A=B=0

T1	T2
S ¹ (A) 1	S ² (B) 1
L ¹ (B) 0	L ² (A) 0

为了证明上述执行不满足顺序一致性，我们构建了图 7-23 所示的执行流图，并根据顺序一致性的规则加入了操作之间的顺序。最终的执行流图中包含了环，这违反了所有内存访问必须存在一个全局一致序的要求，因此，即使这个执行过程满足（纯）一致性，并且对 A 和 B 的访问也符合 store 原子性，但它仍然不满足顺序一致性。

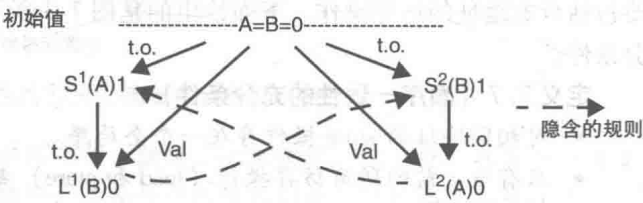


图 7-23 程序对应的执行流图

7.4.3 入站消息管理

在这一节，我们详细探讨顺序一致性系统中的 cache 层次和互连之间的访问缓冲优化机会。我们不考虑本地目录控制器所接受和发送的消息，而是专注于本地 cache 层次所接受和发送的请求/应答。目录控制器接收和发送的流量使用的是不同的数据通路。

图 7-24 给出了一个支持深层存储结构 and 无锁 cache 的系统简化架构模型，我们借助于出站和入站缓冲区来抽象复杂的 cache 层次延迟。注意在处理器核和 cache 之间的 store 缓冲不是图 7-24 所示出站缓冲区的一部分，并且下面的讨论也不会受 store 缓冲的影响，因为我们假定 store 缓冲符合顺序一致性规则。

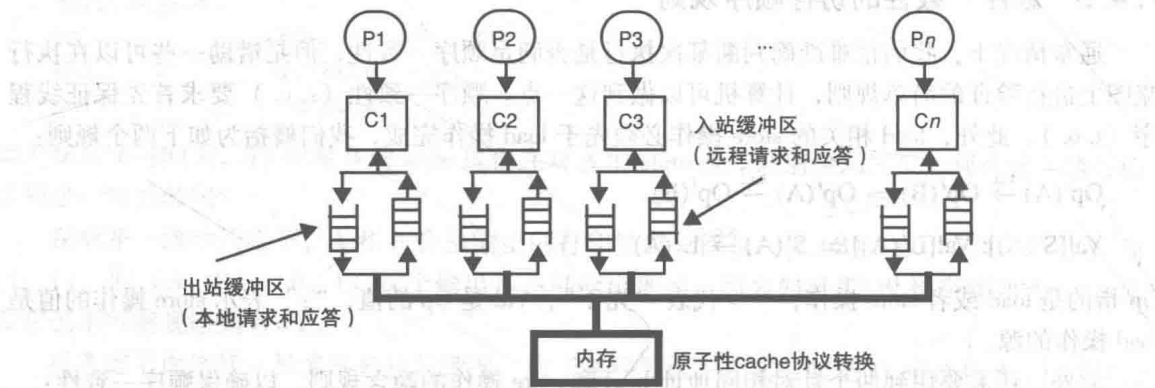


图 7-24 支持入站/出站缓冲区的硬件模型

在支持多级无锁 cache、核内多线程以及多核结构的系统中，可能同时有大量从本地节点发出的请求正在处理，这些请求将和针对输入请求产生的本地响应消息交错进入出站缓冲区，响应也包括从本地脏副本中替换写回的数据，出站请求在 MSI-无效协议时指的是 BusRd 和 BusRdX，在 MSI-更新协议时指的是 BusRd 和 BusUpdate。出站请求的完成由本地 cache 控制器

来控制, cache 控制器收到块副本和应答信号时, 就发出请求完成的信号。在顺序一致性下, 同一时刻每个线程只能有一个这样的出站访存请求 (BusRd, BusRdX, 或者 BusUpdate) 挂起, 因为所有的访存操作是逐个全局完成的。然而, 由于处理器上可能有多个线程在执行, 因此, 出站请求数量可能和线程数一样多。当然, 不同线程的请求必须在所有缓冲区中都被隔离开, 也包括处理器核与 cache 之间的 store 缓冲。

入站消息 (来自远程处理器节点的请求和应答) 同样也需要缓存, 并且处理器节点的入站缓冲区可能包含被不同远程节点请求所触发的多个消息。由于 cache 的复杂层次以及不同层次之间的各种缓冲区的存在, 不同处理器 L1 cache 接收到入站消息的延迟也各不相同。这些入站消息的处理可以在不影响 store 原子性和顺序一致性的前提下进行优化, 目标是减少向目录或总线进行入站消息应答所需要的时间。

参考图 7-24, 理论上在最坏情况下, 入站消息 (在清空、无效或者更新时) 必须遍历节点的整个 cache 层次 (经过所有入站缓冲区), 而消息的应答在发送到主节点之前, 也必须经过所有的出站缓冲区。如果我们能够不通过入站缓冲区和出站缓冲区就完成消息的应答确认, 那么 store 和 load 操作就可以更快全局完成。

幸运的是, 如果入站缓冲区管理得当, 那么许多入站请求都可以在接收到的时候就进行应答确认, 而无需等待来自本地层级 cache 的响应。消息到达 (和应答) 某个节点和它对线程执行产生影响之间的延迟不会影响 store 原子性和顺序一致性。下面以 CC-NUMA 系统为例, 说明在基于目录的 MSI-无效协议和 MSI-更新协议下是如何做到这一点的。

MSI-无效协议

我们采用如示图 7-11 和图 7-16 所示的目录协议, 为了简化说明基本原理, 假设目录控制器在 t_3 时刻安全地将块副本传送给请求者 (T_0), 并且该块副本在 t_4 时刻被接收。因而对外部存储器的访问满足原子性, 并且所有的 store 都全局有序。这里的问题在于, 像 T_1 和 T_2 这样的线程需要对无效请求进行应答, 而根据定义 7.3 和 7.7, 在本地处理器无法返回旧值之后, 无效请求才能被应答。下面将说明, 其实无效请求是可以更快应答的——只要它们达到该节点就可以应答。下面先描述应该怎么做, 然后再给出相应的证明。

在 MSI-无效协议中, 可能会对本地节点产生影响的内站请求主要是无效请求或清空请求, 而输入的应答消息是主节点转发到本地节点的数据块。当本地 cache 中有 Shared (共享) 或者 Modified (修改) 状态的副本时, 需要对输入请求作出反应。而当本地 cache 状态是 Invalid (无效) 时, 则直接丢弃输入请求。

当本地块副本处于 Shared 状态时, 只能接收无效请求。无效请求一旦到达本地节点的内站缓冲区就可以进行应答, 无效请求一旦在内站缓冲区中被接收, store 操作就可以看成是相对本地节点已经完成的。入站的无效请求可能被本地处理器核延后很长时间处理, 当无效请求被应答确认时, 会同时将目录中对应该节点的存在位清零, 因此在目录看来, 该节点上的副本无法再访问到。而如果 cache 接收到某个 Shared 状态块的无效请求而不处理, 那么访问这个块时, 该 cache 实际上就相当于和剩余系统“隔离”开了。一旦隔离开, 本地 cache 也就无法获悉系统中对该块的其他一致性事务处理请求了。

如果本地副本是 Modified 状态, 那么当输入的清空请求到达 cache 层级的内站缓冲区时, 不能直接进行应答确认, 从本地 cache 层级中仍然可以获取到对应的块副本, 同时需先处理在该清空请求之前的所有入站消息。清空请求可能会在本地节点中延迟很长时间处理, 这并不是一个好的解决办法, 因为远程的清空请求者一直在等待该块副本关闭该事务处理。

当本地线程需要发起协议请求处理时 (例如, 需要发送 BusRd 或者 BusRdX 请求), 与该线程相关的之前所有入站消息都必须先在 cache 中生效。因为在 MSI-无效协议中, 每个出站请

求 (BusRd 或 BusRdX) 都被视为 cache 失效并将返回一个块副本, 在失效事务处理对本地线程全局完成之前, 有足够的时间等待数据块返回消息, 并将数据块返回消息之前的所有入站请求在 cache 中生效。

上述优化背后的基本原理是, 共享内存程序的写操作必须和硬件延迟无关。因此, 当本地处理器核收到无效请求或清空请求时, 不需要和当前正在执行的操作进行同步, 事实上, 本地处理器核原本也可能运行得更快, 如果这样那么无效请求或刷新请求就可能在本次执行的晚些时候才能到达该节点。只要本地节点还可以使用自己的本地值 (任何一级 cache 中的都行) 继续执行, 并且入站缓冲区没有发生溢出, 那么当前执行就可以一直继续下去。一旦本地节点往外发送 BusRd 或 BusRdX, 就将与系统的其余部分进行同步。此时, 必须处理完 cache 层次中的所有入站请求。

MSI-更新协议

对于 MSI-更新协议, 尽管原理是一样的, 但是这种情况更为复杂。这是因为如果要保持 store 原子性, 那么在目录向其他副本 (包括请求者) 发送 store_GP 信号进行应答确认之前, 更新信号必须先发出, 应答确认, 并且在本地 cache 中锁定。图 7-17 给出了这一过程, 入站请求包括清空命令、更新命令或 store_GP, 而入站应答是主节点在发生失效时前递的块副本。

当本地副本处于 Shard 状态时, 入站更新操作一旦到达处理器节点就会被应答确认。更新消息会修改该 cache 块并将其锁定, 之后收到 store_GP 消息时再进行解锁, 恢复成可访问状态。当收到上述输入请求时, 可能会延迟较长时间后再进行处理。

当本地节点处于 Modified 状态时, 必须对清空请求进行响应。但是它们不是在收到请求时就立即确认, 而是当请求到达具有有效副本的本地 cache 时, 通过清空该块副本的方式进行确认。这一处理过程同样可能被延迟, 但是请求者会一直等待该副本。在这个处理过程中, 所有先前的入站消息要在 cache 中已经生效。

当本地节点需要发起协议请求处理时 (Update 或 BusRd), 在该事务全局完成之前, 所有的输入消息都必须先被处理。以更新请求为例, 发出更新请求的 cache 会从主节点接收到自己的 store_GP 信号, 然后对 cache 进行解锁, 再关闭该事务 (见图 7-17)。当 cache 接收到自己的 store_GP 信号时, 它必须在对应事务全局完成之前将其处理, 此外, 在这之前的所有入站缓冲区中的消息也都需要处理完。

例 7.11 支持懒惰无效的 Dekker 算法 下面我们一步步详细展示 Dekker 算法。

为了说明新的规则如何应用于入站消息, 我们还是用下面这段相同的代码片段来回顾一下 Dekker 算法:

T1	T2
A=1	B=1
R1=B	R2=A

对应的操作步骤展示在图 7-25 中。

T1 运行在 P1 上, T2 运行在 P2 上。如图 7-25a 所示, 初始时, 两个 cache 都有包含数据块 A 和 B 的共享副本, 并且入站缓冲区是空的。P1 执行 store A 时, 先发出 BusRdX 消息。首先, 会有一个无效请求发送给 P2, P2 一旦收到该无效请求就立即向主节点进行确认, 以便主节点可以将对应的数据块发送给 P1。该数据块由内存返回到 P1 的入站缓冲区中, 而 P2 的入站缓冲区中则会插入一个无效请求。这时, A=1 这个操作相对于 P2 来讲已经完成了 (即使此时对应的无效请求还没有达到 P2 的 cache, 并且 P2 还能读到 A 的旧值), 但是还没有全局完成, 因为输入的数据块副本还没有拷贝到 P1 的 cache 中 (图 7-25b)。

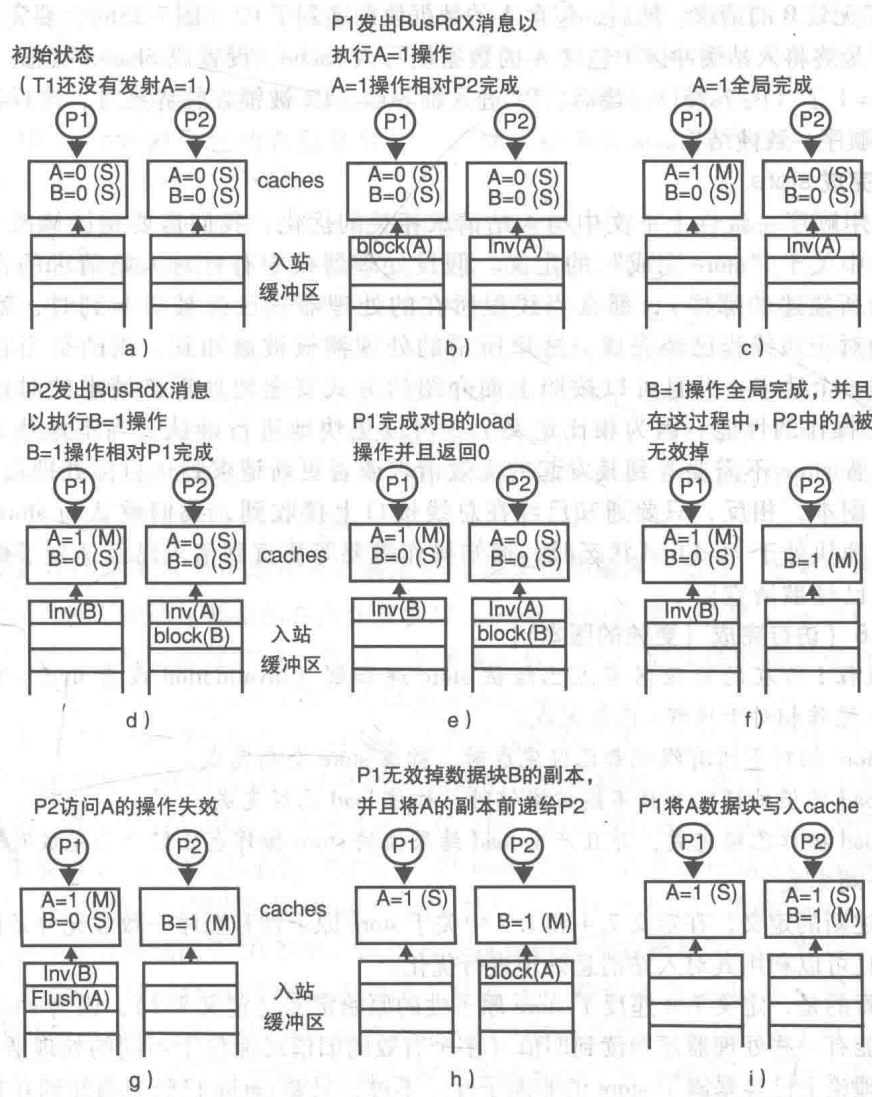


图 7-25 Dekker 算法中的 cache 懒惰无效

在图 7-25c 中，输入块已到达 P1 的 cache 中，并且 P1 可以安全地修改 A 的副本。此时 A=1 就可以认为是全局完成了（尽管 P2 的 cache 还没有被无效掉，它还能读到旧的副本），P1 可以发射对 B 的 load 操作。不过在这之前，P2 在执行 B=1 操作时发出了一个 BusRdX 请求，这导致一个无效请求发送给 P1，缓存在 P1 的入站缓冲区中，并且立即向主节点进行了确认。然后主节点将数据块返回并存入 P2 的入站缓冲区中（图 7-25d）。此时，B=1 操作相对 P1 已经完成，但是仍然不是全局完成，因为该数据块还没有到达 P2 的 cache 中，因此 P2 还无法执行后面的 load。

然后 P1 执行对 B 的 load 操作，并返回 0（仍然忽略掉入站缓冲区中的无效请求）（图 7-25e）。最后，为了继续处理 load 操作，P2 将返回的数据块写入 cache 中，以状态 M 结束 B=1 这一处理过程。在这一过程中，对含有 A 的数据块的无效请求必须在 cache 中生效，然后 P2 将 cache 中 B 的值更新为 1（图 7-25f）。此时，P1 可以读到 B=0，P1 的执行就结束了。而 P2 还有 load A 操作，并且在 cache 中失效了。因此 P2 发出 BusRd，P1 会接到一个清空 A 的入站请求，然后 P2 等待 A 的副本返回（图 7-25g）。最终，P1 决定对清空 A 请求作出响应，响应过程中在

cache 处理了无效 B 的请求。然后, 包含 A 的数据块前递到了 P2 (图 7-25h)。要完成该 load 访问, P2 需要最终将入站缓冲区中包含 A 的数据块写入 cache, 设置成 Shared 状态。此时 P2 就可以读到 $A=1$ 了 (图 7-25i)。最后, P1 进入临界区, P2 被锁在临界区外, 这样我们就得到一个合法的顺序一致性结果。

更快地完成 store

为了利用顺序一致性上下文中与入站请求相关的优化, 我们需要稍微修改一下 7.3.3 节定义 7.3 中关于“store 完成”的定义。假设处理器核中有针对入站请求的合理管理机制 (如上面所描述的那样), 那么当线程所在的处理器核已经被通知到时, 就可以认为 store 操作相对于该线程已经完成。这里所谓的处理器核被通知到, 指的是当它在入站缓冲区中收到一个请求, 并且可以按照上面介绍的方式安全地处理该请求的时候。这有助于提高 store 操作的性能, 因为相比定义 7.3 可以更快地进行确认。当本地块处于 Shared 状态时, 远程 store 不需要等到其发起的无效请求或者更新请求到达目标处理器核 cache 层次中的所有副本, 相反, 只要通知已经在总线接口上接收到, 我们就认为 store 操作已经完成。当本地块处于 Modified 状态时, 通知操作需要等待直到请求已经达到了修改的副本并且该数据已经被清空后。

定义 7.8 (访存完成 (更快的版本))

- 当线程 i 所在的处理器节点已经被 store 通知到 (invalidation 或者 update) 时, 称该 store 操作相对于线程 i 已经完成。
- 当 store 相对于所有线程都已经完成时, 称该 store 全局完成。
- 当 load 的值被锁定并且不能被撤销时, 称该 load 已经完成。
- 当 load 操作已经完成, 并且产生 load 结果值的 store 操作也已经全局完成时, 称 load 操作全局完成。

基于上述新的定义, 在定义 7.4 和 7.7 中关于 store 原子性和顺序一致性充分条件的约定仍然成立, 并且可以利用其对入站消息处理进行优化。

值得注意的是, 定义 7.8 违反了 store 原子性的原始定义 (定义 7.2), 因为 store 全局完成时, 仍然可能有一些处理器还能读到旧值 (多个有效的旧值可能位于不同的处理器 cache 中), 这意味着在理论上已经暴露了 store 的非原子性。不过, 只要 cache 已经被通知到并且在每个节点内部小心处理, 那么软件就无法检测到对原子性的违背。

图 7-25c 中可以清晰地看到对于 store 原子性基本定义的违背, 根据定义 7.8, 在 P1 中的 $A=1$ 这个 store 操作全局完成时, 同时存在 A 的两个不同副本, 并且 P2 的 load 还可以返回 $A=0$ 。根据新的充分条件的定义, P1 和 P2 cache 中的 A 的值都是 GP 并且可访问的, 但它们的值却是不同的。然而, 根据新的充分条件定义, 这个执行是满足顺序一致性的, 即使 P2 执行 load A 操作仍然返回 0, 这个执行过程也还是满足顺序一致性的。

基于运行时间的不可预测性, 我们可以获得很多优化机会, 每个节点中入站消息可能的重排序就是一个例子。比如, 在 MSI-无效协议中, 连续的多个入站消息可能会被乱序处理, 但这并不违反顺序一致性, 因为要想观测到无效数据就需要发起一次失效请求, 而这将导致入站缓冲区中的所有消息都被处理。这些优化策略很有价值, 但是详细介绍则需要很大篇幅, 完全可以重新写一本书了, 因此, 在本书中, 我们将不做具体讨论。

7.4.4 store 同步性

store 同步性是从另一个角度来观察时序对于 store 原子性和顺序一致性的影响, store 同步性定义如下:

定义 7.9 (store 同步性) 如果对所有地址的所有 store 操作都存在一个全局序, 并且任意两个线程观察到的 store 顺序都是相同的, 那么称该存储系统满足 store 同步性。

满足 store 同步性的存储系统和满足 store 原子性的存储系统是等价的。

定义 7.10 (store 原子性的充要条件) 存储系统满足 store 原子性, 当且仅当它满足 store 同步性。

从硬件层面来看, store 原子性和 store 同步性之间的区别在于, 在 store 同步性中, 不同线程在同一时刻可以观察到同一位置的不同值, 因此 store 同步性违反了定义 7.1 ~ 定义 7.4, 不过, 软件察觉不到这一点, 因为软件的实现是不依赖于任何特定时序的。而 store 同步性也正是利用了这样一个事实: store 操作可能以不同速度传播到所有处理器节点, 软件实现上必须与具体时间无关。对软件来讲真正重要的是它所观察到的对所有地址的所有 store 操作的全局顺序, 而不是绝对时间。注意, 如果所有处理器都按照线程顺序挨个执行所有的 load 和 store, 那么满足 store 原子性的系统也满足顺序一致性, 因为所有 store 存在一个全局序, 并且 load 只能按照线程顺序、而不能乱序观察到 store 的值。

例 7.12 支持前递的 store 缓冲可以满足 store 原子性和顺序一致性 如果考虑所有内存地址的访问, 那么就可能通过调度访存操作来实现 store 缓冲的前递, 这样 store 操作在软件看来就是原子的, 整个过程也满足顺序一致性。下面我们根据 store 同步性的定义进行展示, 整个例子说明了如何利用 store 同步性来解释存储系统满足 store 原子性。

存储系统的结构如图 7-19 所示, 存储系统假定满足原子性, 但是 load 可以从本地 store 缓冲中返回值。这和图 7-20 中的 load 以及 store 调度的主要区别在于, 我们现在考虑了不同内存地址之间的相互作用, 并且跟踪所有地址的所有访问, 而不仅仅是一个地址。如图 7-20 所示, load 可以从本地 store 缓冲中返回值, 然而, 一旦 load 没有在本地的 store 缓冲中找到对应值, 那么 store 缓冲中的内容就必须先传播到 cache 中, 然后 load 操作才可以访问 cache。针对所有地址访问进行定序的过程和图 7-20 类似, 区别在于这里包括所有地址访问。当 load 没有在本地的 store 缓冲中命中时, 通过将 store 缓冲中的所有本地 store 都传播到 cache 中, 就可以将之前所有在 store 缓冲中已经完成的访问 (load 和 store) 插入到包括所有地址的 load 和 store 操作的全局序中。由于这些 store 还没有被除了本地线程之外的其他线程观察到, 因此所有在 store 缓冲项中本地执行的 load 和 store 可以用打包插入的方式来构建 load 和 store 操作的全局序, 就像我们之前针对单地址所做的那样。如果访存操作按照线程顺序提交到 store 缓冲, 那么该系统依然满足顺序一致性, 因为在所有访问的全局序中, load 和 store 的顺序也符合线程顺序, 因此满足 7.4.1 节中的条件。顺序一致性系统对 load 的约束使得整个存储系统 (包括 store 缓冲) 都满足 store 原子性。

图 7-26 给出了 store 原子性、store 同步性 (从软件角度看, 与 store 原子性是一样的)、纯一致性以及无序内存之间的差别。我们为各种系统绘制出“值可观测线”。这些线显示了处理器可以观测到新值的最早可能 (实际) 时间, 以及处理器有机会通过执行 load 操作读到新值的时间。针对两个不同地址 X 和 Y 的可观测线, 如图 7-26 所示。

在满足严格一致性的存储系统中, store 的值同时对所有线程可见。在 store 同步系统中 (和 store 原子性系统一样), store 值可能不同的时间对不同处理器可见, 但是值可观测线不会有交叉, 所有处理器以相同的顺序观察到 store 的值。在硬件层面会违反 store 原子性 (处理器在相同时刻可以观察到不同 store 的值), 然而时序无关的软件无法检测到这种硬件层面的非原子性, 因此我们仍然认为该存储系统满足原子性。图 7-26b 和图 7-26c 展示了 store 原子性和纯一致性之间的区别, 在纯一致性下, 不同 store 处理器可以在相同时刻观察到相同位置的不

同值，而且 store 值的可观测线可能会交叉，使得对不同地址的 store 可能被不同线程按不同的顺序观察到。最后，图 7-26c 和图 7-26d 说明了纯一致性和无序内存（不符合一致性）之间的差别，在无序内存中，两个线程可以按不同顺序观察到对同一内存地址的两次 store。

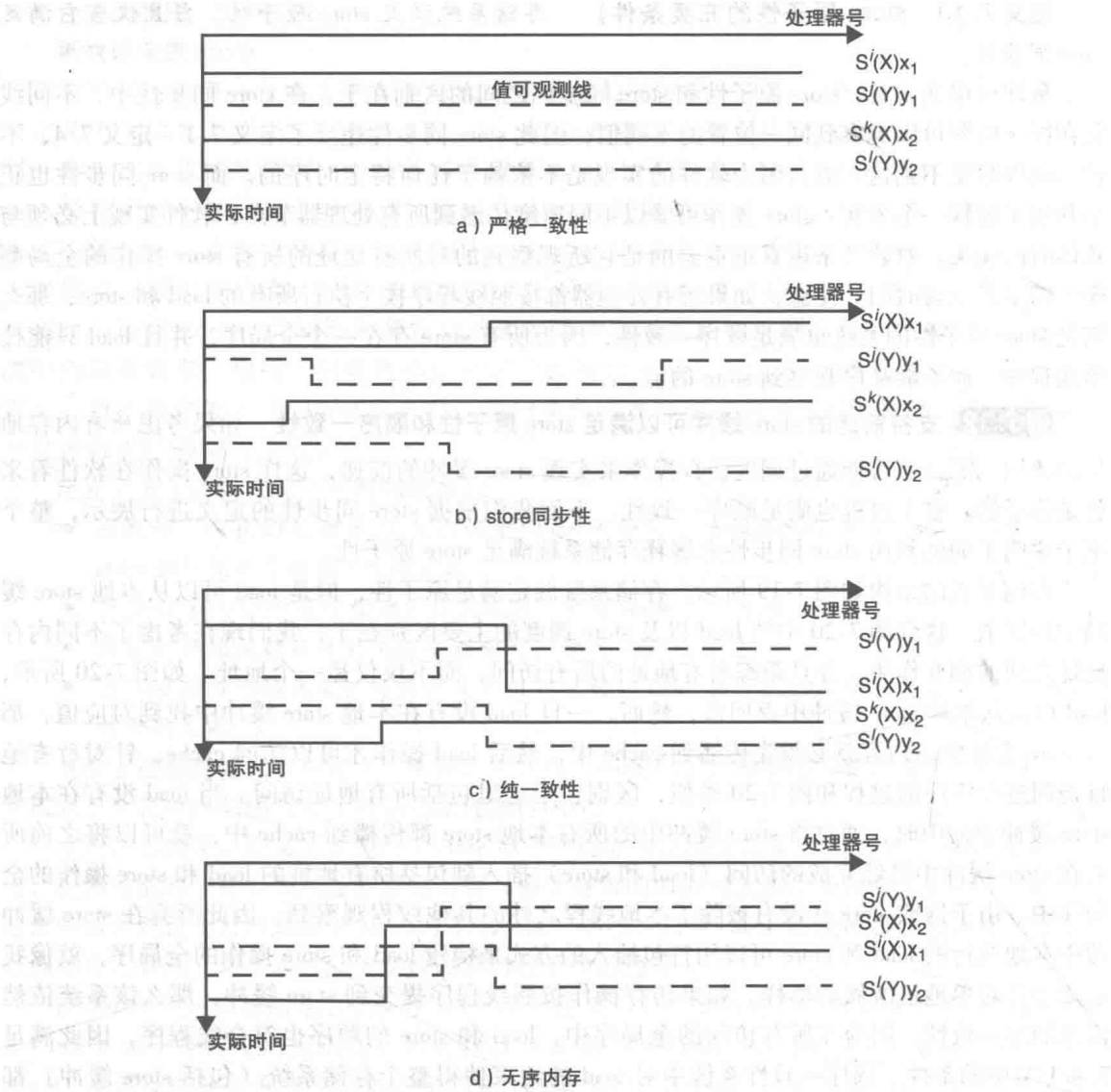


图 7-26 值可观测线

7.5 同步

在多任务程序中，对线程间进行可靠同步的需求远比一致性（coherence）或者存储一致性模型更重要，即使系统满足顺序一致性，也仍然需要同步支持。同步是计算机中一个非常古老的问题，可以追溯到多任务操作系统的问世，早在多处理器系统出现之前，同步原语的正确执行就已经是一个基本的系统需求。

20 世纪 60 年代，当时电脑使用的还是基于打孔卡的编程，计算任务也是以批处理形式提交，从那时开始，用户就通过分时的方式共享一些计算机资源，比如处理器、存储器和 I/O 设备等。分时操作系统必须正确、可靠、公平地进行资源共享。用户在时间上复用这些资源，在

保证合理的个人响应时间的情况下最大化系统吞吐量。即使现在，共享和对系统资源的分时复用仍然是高端服务器的重要目标，因为资源的成本都很高，并且单个用户难以充分利用所有资源。通常情况下，用户进程在 CPU 上连续运行，直到它请求操作系统服务来访问共享资源时才会中断。每个进程都会分配一个固定的时间片——最长周期数，当时间片结束时，进程就会被抢占，这样其他用户进程可以获得机会运行。基于分时共享的单处理器系统现在通常称为软件多线程。

分时共享单处理器系统的多个进程也可以共享一些内存位置，当进程共享内存时，需要通过同步来获得共享内存地址的访问权。当多个处理器核共享内存的时候，同步就更加关键，因为这种情况下的线程或进程是并发执行的，而不是一次只有一个。

7.5.1 基本同步原语

当共享物理资源（例如打印机或共享内存区域）可被多个代理（进程或线程）访问时，任意时刻只能有一个代理同时访问。在分时操作系统中，这一需求可以通过同步标志或内存中的信号量来保证。共享变量指示了资源是否处于忙的状态，对这类共享变量的更新在计算机体系结构和操作系统中是一个非常关键而又基本的问题。

基本的锁问题

下面我们介绍在多线程中共享内存位置的一些最基本问题。线程在访问共享可写变量时必须进行互斥，这意味着任何时间只有一个线程可以访问该变量。

假设下面修改共享变量 A 的语句在两个不同的线程（T1 和 T2）中执行（下面给出的是仅有的两条动态更新 A 的语句）：

```
T1:      T2:
...      ...
A=A+1    A=A+1
...      ...
```

程序员期望的结果是：无论这两条语句以什么顺序执行，最终的结果都是 A + 2。但是这种直觉是不可靠的，因为这个加 1 的语句可能不是原子执行的，编译后的代码实际上可能包含多条指令，例如，load/store 和算术/逻辑指令。

单处理器上一次只能执行一个线程，可能是按照下列指令顺序执行（具体取决于操作系统特定的调度机制）：

```
T1:      LW R1,A           /T1 IS PREEMPTED
.....
T2:      LW R1,A           /LATER, T2 RUNS
T2:      ADDI R1,R1,1
T2:      SW R1,A
..... /T2 IS PREEMPTED
T1:      ADDI R1,R1,1      /T1 RESUMES
T1:      SW R1,A
```

T1 先执行 load，然后时间片用完后切换到 T2 调度执行，执行三条指令后也被抢占出去，然后继续调度 T1 执行并执行完剩余的指令。可以看到，最终的结果是 A + 1 而不是程序员所期望的 A + 2。

在多处理器系统（或硬件多线程处理器核）中，线程 T1 和 T2 可能在不同的线程上下文或者处理器核上并发执行，它们可能同时在执行 A + 1 的操作，在这种情况下，可能出现下面的指令动态交错情况（假设顺序一致性）：

```

T1      T2
...
LW R1,A
        LW R1,A
ADDI R1,R1,1
        ADDI R1,R1,1
SW R1,A
        SW R1,A

```

同样，这次的结果还是 $A=1$ ，而不是预期的 $A=2$ 。如果想确保 A 的最终值总是 2，那么 $A+1$ 这条语句的执行过程必须满足原子性，或者是不同线程在执行这条语句时在时间上不会有任何的重叠（互斥）。上述经典问题可以用临界区来解决，临界区包括一个受保护的、可被多个线程执行的代码段，代码段中包含对共享数据的读/写访问，临界区内的代码段在任何时间只能在一个线程中执行。

在支持软件多线程且只有一个执行上下文的单处理器中，临界区代码可以通过关闭中断的方式来实现：

```

T1
...
disable interrupts
A=A+1
enable interrupts
...

```

然而，在硬件多线程处理器核或者多核系统中，关闭中断是没有用的，因为线程是在不同的上下文中并行执行的。这种情况下必须使用锁。锁是一种用来保护临界区内关键代码段的原语。典型的锁是一个二进制标志位，当值为 0 时，表示锁可用。线程通过将锁设置为 1 来获取（acquire）锁，当锁被设置时，获得该锁的线程可以继续执行临界区的代码，而其他线程被限制在同一把锁所对应的临界区外，等待锁释放（release）。处于临界区内的线程可以随意地读写共享变量。在临界区代码的最后，线程通过将锁的值重置为 0 来释放锁（这通常称为解锁或释放操作）。当锁被释放时，临界区内发生的修改对其他线程可见：

```

T1      T2
...
Lock(La)  Lock(La)
A=A+1     A=A+1
unlock(La) unlock(La)
...

```

锁可以通过对共享内存标志位的简单 load 和 store 操作实现，这里我们以 Dekker（或 Peterson）算法为例，其最简单的版本包括两个线程：

```

INIT A=B=0
T1      T2
...
A=1     B=1 /acquire
while(B==1); while(A==1);
<critical section> <critical section>
A=0     B=0 /release

```

T1 对标志 A 置位，以此告诉 T2 它想进入临界区代码；同样，T2 对标志 B 进行置位。然后 T1 检查标志 B 以验证 T2 的情况，如果标志 B 被置位，那么 T1 等待。这段代码确保某一时刻

间至多只有一个线程能执行临界区代码，从而表现出锁的作用。在临界区的最后，会对阻塞其他线程的标志位进行复位。这种使用常规共享变量实现锁的方法有如下几个缺陷：

- 如果两个线程同时设置了标志位，那么会出现死锁（当然我们可以通过更复杂的代码来解决）；
- 当线程数增加时，代码会变得很复杂；
- 只有在顺序一致性下，上述代码才能正确工作。

由于上述原因，我们通常使用特殊的硬件支持来实现锁机制，比如原子的 RMW（read-modify-write）指令（test_and_set），专门的总线，或者同步寄存器等。

barrier（栅栏）

barrier 是多个线程之间的同步协议，只有当所有线程都到达 barrier 时，才能继续执行 barrier 后的操作。下面以两个线程为例给出一个简单的 barrier 代码：

```
INIT BAR=0
T1                T2
...
Lock(bar_lock);   Lock(bar_lock);
BAR= BAR+1;        BAR= BAR +1;
Unlock(bar_lock);  Unlock(bar_lock);
while (BAR < 2);    while (BAR < 2);
```

两个线程都在 barrier 处将 barrier 计数器（BAR）分别加 1，然后检测 barrier 值，等待直到 BAR 变为 2，然后再继续执行。注意，对 barrier 计数器的加 1 操作必须在临界区完成，但是 while 循环中的读 BAR 操作没必要放到临界区内，因为 BAR 的值是单调增加的，并且 while 循环要检测的是 BAR 到达最大值的时刻。这个例子说明要想高效而正确地编写共享内存程序是多么困难。

barrier 在用循环实现的迭代算法中广泛使用，图 7-2 展示了一个 Jacobi 迭代算法的例子。每个迭代包含两步：在第一步中，线程按互斥方式修改和访问 X_i 的值；在第二步中，所有线程按只读方式访问 X_i 的值。对 Y_i 值的访问也是类似。算法通过 barrier 将不同的迭代以及迭代内的不同步骤之间进行了区分。在通常情况下，barrier 可以用于实现有大量数据访问的临界区，并且比简单对大量数据加锁的方法更加高效。

还有一个问题是，例子中的 barrier 代码必须在后续迭代中重复执行。因此，上面这种简单 barrier 实现的代码必须进行修改，在每次 barrier 完成之后可以复位。这也是个不小的问题，目前这份简单代码中总是假定 BAR 是单调增加的。

点对点（生产者/消费者）同步

有时候生产者线程需要向消费者线程发送信号，告诉对方自己已经执行到某个位置，这种情况可以通过内存中的一个简单共享标志位来实现：

```
INIT A=FLAG=0
T1                T2
...
A=1;
FLAG=1;           /release
while (FLAG==0);   /acquire
print A
```

在本例中，T2 是生产者，T1 是消费者，且 T1 中打印的 A 的值必须是 1。对 FLAG 的访问不需要临界区保护，因为只有一个线程可以修改 FLAG。在单个生产者、多个消费者的情况下

也是同样的。这段代码的问题在于它依赖于特定的存储一致性模型，并且可能不适用于某些存储一致性模型。

7.5.2 基于硬件的同步

锁和 barrier 可以通过专门的硬件来实现，比如总线或者寄存器/触发器。barrier 可以通过使用开集互连的专门总线来实现，初始状态为高。当线程检查 barrier 时，就会试图将其拉低，只有当所有线程都这样做时，操作才能成功，此时所有线程都已经到达 barrier 点。

共享同步寄存器可以实现同样的功能，区别在于寄存器可以在多个时钟周期内保存值（类似于内存），而总线不能（总线只能在当前时钟保持值）。此外，还可以用共享触发器来实现锁，用共享计数寄存器来实现 barrier。

硬件实现同步原语的缺陷如下：

- 可扩展性差。需要用到内存之外的其他共享资源，而共享资源的带宽制约了其所能连接的处理器数目。
- 灵活性有限。如果硬件同步资源（总线、触发器、寄存器）不够，那么当硬件资源达到最大数时，要么将某些线程暂停，要么对同步硬件资源进行分时复用（虚拟化），这比用共享内存来实施同步的方法要更加困难，效率也更低。
- 复杂度更高。现在的处理器都是多线程的，因此需要在每个线程（而不仅仅是每个处理器）上都做硬件的同步支持。

相比之下，共享内存可以为同步锁提供足够的共享内存地址，而 cache 机制也有助于提高内存访问的可扩展性，包括对诸如锁这样的共享同步数据的访问。

7.5.3 基于软件的同步

从历史上看，同步大多是通过共享内存和 RMW（read-modify-write）这样的特殊共享内存操作指令实现的。RMW 指令是 load 和 store 之外的一类新的访存指令，RMW 指令从内存读值，进行修改，然后再将新值写回内存，整个过程是原子完成的，所有的现代 ISA 都有提供支持 RMW 原子操作的指令。

借助于原子访存指令，可以实现可靠而复杂的锁机制。为了说明 RMW 指令在共享内存锁机制中的必要性，下面先给出一个简单使用 load 和 store 指令加锁的方法：

```
Lock:    LW R2,lock      /
          BNEZ R2,Lock
          SW R1,lock      /R1 = 1
Unlock:  SW R0,lock
```

在上面的代码中，锁一直处于“忙等”状态，直到返回值为 0，然后再将 lock 设置为 1。加锁是为了解决原子性的问题，但是加锁代码本身也面临同样的问题——缺乏原子性保证。如果两个线程同时将值为 0 的 lock 加载到 R2 寄存器中，那么将导致两个线程同时进入临界区，这样会出现意想不到的结果。

Test_and_set 指令

诸如 test_and_set（T&S）这样的原子 RMW 指令解决了锁的问题。T&S 指令可以读取内存位置，同时原子性将其设置为 1，并将读到的值返回寄存器：

```
T&S R1,lock
```

如果 R1 中返回的值是 0（成功），则获得锁；如果 R1 中返回的值是 1，则表示获取锁失败。使用 T&S 指令，可以在多处理器系统中实现锁机制。

```
Lock:    T&S R1,lock
        BNEZ R1,Lock
```

```
Unlock: SW R0,lock
```

T&S 指令是除 load 和 store 之外的一类新的访存指令，软件总是认为其执行过程是原子的。如果锁不可缓存，那么这个操作可以在内存中完成，如果锁可以缓存，那么在 cache 中完成。如果 T&S 在内存中执行，那么将绕过 cache，这时内存控制器可以先执行一条 load，紧接着执行一个 store 1 的操作，并将 load 的值返回，这样就可以保证 RMW 的原子性。如果 T&S 在 cache 中执行，那么对应的 cache 协议应该是一个基于无效机制的协议，比如 MSI-无效协议。在 MSI-无效协议中，T&S 被当作 store 来处理，因此在尝试执行 T&S 之前，线程的 cache 必须获得一份 Modified 状态的唯一副本。一旦获得 Modified 状态副本，T&S 就可以在数据块被清空之前通过 cache 控制器在 cache 中执行。如果处理器核支持多线程，并且 L1 cache 中对应锁的副本是 Modified 状态，那么同一个核上的多个线程可能会竞争 L1 cache 中的同一把锁。

无论使用 MSI-无效协议的 T&S 指令是在内存中还是在 cache 中执行，上述加锁的代码都会在存储总线上产生大量流量：没有 cache 的情况下，所有 T&S 都必须到达内存执行；而在有 cache 的情况下，当不同核的线程都处于等待锁的状态时，也会导致包含锁的数据块在不同核的 cache 之间来回传递。下面假定锁在 cache 中执行，现代系统中一般也都是这样的。

为了减少 cache 之间的通信量，处于循环忙等状态的 T&S 可以先暂停一段时间后再重试。在诸如以太网的网络协议中，通常使用指数回退算法来解决这一问题，每次 T&S 失败时，到下次尝试之间的延迟按照指数增加，也就是说，第 i 次尝试时的回退延迟是 $k \times c^i$ 。出于这个目的，每次失败后都会执行一个空循环。

利用 cache 协议的一个常用技术是“test_and_test&set”锁：

```
Lock:    LW R1,lock
        BNEZ R1,Lock
        T&S R1,lock
        BNEZ R1,Lock
```

```
Unlock: SW R0,lock
```

这段代码中，忙等循环先执行一个常规的 load，并且在 MSI-无效协议的 cache 中命中，只要 load 返回 0，T&S 就试图执行原子操作抢占锁。如果 T&S 失败（如果多个线程同时获取同一把锁，这种情况就可能发生），那么又重新回到执行 load 的忙等循环的开始位置。当锁处于忙状态，并且存在多个线程争夺锁时，test_and_test&set 锁机制可以减少访存通信量。

其他的 RMW 指令

除了 test_and_set 之外，在现代指令集中还包括其他 RMW 指令。

- swap 对 test_and_set 进行了扩展，存在寄存器中的 lock 值可以设置成任意值：

```
SWAP Rx,lock
```

Rx 中的值和内存 lock 处的值进行原子交换。当寄存器中的值是 1 时，就对应 test_and_set，因此 test_and_set 是 swap 的一种特殊情况。如果交换中的寄存器使用 R0，swap 也可以用来进行解锁。

- compare_and_swap (CAS) 类似于 swap，区别在于只有当条件满足才会执行交换操作：

```
CAS Rx,Ry,lock
```

比较 Rx 和内存地址 lock 处的值，如果相等，将 Ry 和 lock 中的值进行交换（整个过程原

子完成)。CAS 常用于管理队列的原子插入和删除。

- `fetch_and_op` (F&OP) 返回一个内存位置的值, 然后对该内存位置执行 OP 操作, 整个过程原子完成。F&OP 的最常见形式是 F&ADD:

```
F&ADD Rx, lock, Imm
```

内存地址 `lock` 处的值读取到 `Rx` 中, 然后 `lock` 和立即数 `Imm` 相加, 整个过程原子完成。在做动态循环迭代调度时, F&ADD 是一条非常有用的指令。

Load locked (LL) 和 store conditional (SC)

RMW 指令属于复杂指令, 不是很适合五级流水线或者乱序执行的 RISC 流水线, 因为该指令要求原子执行两个访存操作 (一次 `load` 和一次 `store`)。实际上, RMW 指令中的 `load` 和 `store` 也可以是单独的指令, 只要硬件能确保在这两条指令之间的整个执行过程不会违反原子性。使用两条单独的指令来实现 T&S 锁的方法如下所示:

```
T&S(Rx, lock):  ADDI R1, R0, 1
                LL Rx, lock
                SC R1, lock
                BEQZ R1, T&S
                return
```

LL 指令 (Load-linked 或者 load-locked) 将内存地址 `lock` 处的值加载到 `Rx`, SC (store conditional) 只有在 LL 之后没有其他 `store` 执行时, 才会更新内存。否则, 将取消对内存的更新, `R1` 寄存器返回 0。如果 `R1` 返回的是 0, 那么接着重试 LL。一旦 SC 返回一个非零值到 `R1`, 上述 test-and-set 操作成功将 `lock` 的值返回到 `Rx`。在支持条件标志的机器中, 可以使用一个条件位来表示 SC 执行失败。

除了比较适合 RISC 和带推测的 OoO 流水线外, 基于 LL 和 SC 的 RMW 实现也非常灵活。我们可以通过简单改变 LL 和 SC 之间的代码, 就能实现各种复杂多样的 RMW 原语操作。当然也会有一些限制, 比如 LL 和 SC 之间的代码越复杂, SC 就越可能会失败。此外, LL 和 SC 之间的代码应当避免出现不可撤销的操作, 比如 `store` 指令或者可能造成异常的指令。

LL/SC 的实现依赖于 cache 一致性协议, 监听单元或者网络接口上支持一个 LL 位, 执行 LL 指令时会对 LL 位进行置位。监听单元或网络接口对输入信号进行侦听, 一旦接收到无效或者更新消息时会对 LL 位进行复位, 从而导致随后的 SC 执行失败。网络接口上可以支持多个标识了锁地址的 LL 位, LL 位由 SC 进行复位。

例 7.13 使用 LL 和 SC 实现 CAS 给出 CAS (`Rx`, `Ry`, `X`) 的实现代码, `Rx` 的值先和内存地址 `X` 的值进行比较, 如果相等, 将 `Ry` 和 `X` 中的值进行交换, 否则 `Ry` 保持不变。

```
CAS(Rx, Ry, X)  ADD R2, Ry, R0    /save Ry in R2
                LL R1, X
                BNE Rx, R1, return
                SC R2, X            /attempt to store Ry
                BEQZ R2, CAS
                ADD Ry, R1, R0      /return X in Ry
                return:
```

在上述代码中, 需要在多个 CAS 迭代中分别保存 `Ry` 的值。如果 SC 执行成功或者失败的结果可以通过一个条件位来表示, 那么就不需要每个迭代都保持 `Ry` 了。

信号量

信号量 (semaphore) 是建立在加锁和解锁原语之上的系统级同步原语。信号量包含两个

操作 P 和 V: P(sem) 是用来获取信号量的内核调用 (进入临界区); V(sem) 是释放信号量的内核调用 (退出临界区)。

布尔信号量是一个行为类似于锁的二进制变量, 值为 1 时, 表示该信号处于释放状态, 值为 0 时, 该信号量处于忙状态。布尔信号量通常用来控制大型临界区的访问, 临界区要大到足以支撑线程抢占和调度的开销。

```
P(sem): if (sem>0) then sem--  
        else <block calling thread and switch to another thread>;  
  
V(sem): if <there is a thread waiting on sem> then  
        <select waiting thread and wake it up>  
        else sem++;
```

信号量 (semaphore) 先初始化为 1, 如果执行 P 操作时信号量处于忙状态, 那么线程将被挂起, 其对应描述符插入到与该信号量关联的等待队列中, 然后启动其他线程执行。执行 V 操作时, 将查询与该信号量关联的等待队列, 如果队列为空, 则释放信号量, 否则从等待队列中激活具有最高优先级的线程执行。

同步事件的构成

线程上的软件同步原语或操作 (如 barrier, thread_create 或 thread_terminate) 是建立在锁维护的临界区的基础上的。锁必须先执行 acquire (获取) 后执行 release (释放)。acquire 是为了获得同步变量的访问权, 以便执行同步点后的操作。而 release 则是为了使其他线程也可以通过同步点, release 可以通过简单的 store R0 或者 swap R0 操作来完成, acquire 的实现则更加复杂, 需要包含相关的等待机制。

一种常见的等待机制是忙等 (busy waiting), 当锁处于忙状态时, 线程会一直对锁进行检测, 直到解锁。上述过程会消耗大量处理器资源和内存带宽, 通过将线程调度出去可以减轻这种开销。在硬件多线程处理器核中, 等待锁的线程可能被暂停, 或者运行优先级可能被降低。在某些情况下, 锁等待可能会导致某些正在运行的线程 (块) 被挂起, 并从 ready-to-run 线程列表中删除。在某些机器上, 这可能意味着需要收回之前分配的线程上下文。

另一种等待方法是阻塞 (blocking)。在支持信号量的操作系统中, P 就是一个阻塞操作。当执行 acquire 操作时, 如果信号量忙, 那么线程就会取消调度, 并将其放入与其关联的信号量等待队列中, 然后调度另一个线程到硬件线程上下文中, 从而将处理器释放给其他线程执行。

选择哪种等待方法 (忙等还是阻塞) 主要取决于预期的等待时间。忙等方式几乎没有线程调度开销, 但是会消耗硬件资源。因此, 只能用于预计等待时间很短的情况。相反, 如果预计的等待时间很长, 那么最好是抢占该等待线程, 将硬件上下文分配给其他线程。另外, 在某些情况下, 可能混合方法会更有效: 首先, 线程试图通过忙等来获得锁, 如果试验几次不成功的话, 再将该线程挂起到等待队列中。

如果系统中除了正在等待的线程外没有其他可运行线程, 那么不管需要等多久, 都可以采用忙等策略。最后, 在操作系统内核级别, 忙等通常是必需的一种策略, 因为内核直接和裸的硬件交互, 在这个层次上没有任何关于抢占和线程调度的硬件机制支持。

7.6 放松的存储一致性模型

顺序一致性是程序员所期望的最严格的一种存储一致性模型。在顺序一致性模型中, 不同线程的所有访存指令按照全局一致序交错执行, 并且线程内部满足线程序。当然, 我们还可以

store 的值)，其对应的传播过程就可以用 store 缓冲来建模。比如，线程私有的 L1 cache 可以是无锁的，当发生 store 失效时，可能在数据块返回之前就在 cache 中分配了一个 cache 行，此时 store 就全局完成了。在这种情况下，所有对该未完成数据块执行的 store 操作都被看成是线程 store 流水线的一部分，以及模型 store 缓冲中的一部分。

本地 store 流水线中的 store 是否会将它们的值前递给接下来的 load 操作，这对性能和正确性都有影响。支持前递时，load 可能在对应 store 操作全局完成之前就从本地 store 流水线中返回对应的值。因此，在所有共享内存操作组成的全局序中，load 会越过 store。通常对所有内存位置的所有访问无法形成一个一致性顺序，因为大多数执行中，纯一致性和不同地址的访问顺序之间都会产生冲突。相反，如果不支持前递功能，load 必须先等待前面相同地址的 store 已经在模型的 store 缓冲中完成（这意味着大家都可以看到 store 的值）之后，才能在 cache 中执行。

从硬件的角度可以看到，图 7-27 硬件模型中的访存执行必须符合每个线程中对全局完成顺序的要求。图 7-28a 中给出了这些顺序要求，需要确保的主要是 4 种顺序：load-load，load-store，store-load，store-store。两个访问之间的箭头表示在所有前面的第一类访问已经全局完成之前，第二类访问不能发射到 cache。例如，store-to-load 顺序表示在所有前面的（按照线程程序在前面的）store 都已经全局完成之前，不会将 load 发射到 cache 中执行。

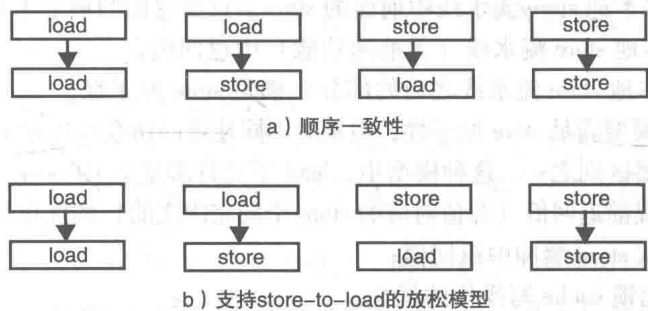


图 7-28 以硬件为中心的存储一致性模型中需确保的顺序

定义 7.11（符合存储一致性模型） 如果给定架构下的所有可能的执行在某种存储一致性模型看来都是有效的，那么我们称这种架构符合该存储一致性模型。

上述条件可能会在测试过程中用到，不过，要想证明某个架构符合给定的模型，还必须对硬件进行约束，以便找到一种系统方法将目标系统上所有可能的执行过程都映射到模型上。常见的约束条件是要求 cache/内存访问满足原子性。比如，可以利用定义 7.4 和定义 7.8 来实现访存的原子性。接下来，我们将探讨存储一致性模型的主要特征，这跟 load 和 store 的顺序有关。主要考虑两类模型：顺序一致性模型和支持 store-to-load 放松的模型。

顺序一致性

在顺序一致性模型下，所有访存操作构成的全局序也必须同时满足所有线程内部的线程序，如图 7-28a 所示。前面我们基于 store 同步性在例 7.12 中实现了优化。在图 7-3 所示的架构中，所有的顺序性可以通过如下方式保证：

- load-store 和 load-load：load 操作阻塞在访存流水级（ME），并且在 load 值返回之前，会一直阻塞住流水线。当某个 load 在 ME 流水级阻塞时，所有后续的 load 和 store 都会被阻塞住。
- store-load：load 的值可以从本地 store 流水线中返回。但是如果 load 的地址不在本地 store 流水线中，则 load 必须等到 store 缓冲中前面所有的 store 都已经在 cache 中全局完成后才能继续执行。

- store-store: store 缓冲中的 store 操作必须以 FIFO 顺序逐个全局完成。由于 store-store 顺序的约束, 不能对 store 缓冲项进行合并, 除非合并的两个 store 之间没有任何对其他地址的 store 操作。

顺序一致性中的 store-load 顺序约束使得 store 缓冲几乎发挥不了什么作用, 而 store 缓冲的高效利用对流水线处理器的性能又非常关键, 因此, 很多存储一致性模型对 store 和后续 load 之间的顺序约束进行了放松, 如图 7-28b 所示。不管支不支持前递机制, 对 store-load 的放松都可以提高 store 缓冲的效率。比如, 当 store 在 cache 中失效时, 随后的 load 可以继续访问 cache, 而此时前面的 store 仍然在等待。

由于还有 store-store 和 load-load 的顺序约束, 对 store-load 顺序放松之后的系统只满足如下特征: 线程的 store 不能被其他线程以不同顺序观察到。在这种机器上, 某些用于保证顺序一致性的代码 (比如使用二进制标志进行点对点通信) 还能够正确运行, 因为每个线程的 store 必须被其他线程以线程的方式看到。但是其他一些基于顺序一致性系统实现的代码 (如 Dekker 算法) 就无法正确运行了, 因为 load 可以越过前面的 store。

不支持前递机制的 Store-to-load 放松模型

这种模型中没有哪个 load 可以返回一个非 GP 值就执行完成, 模型的基本规则如下:

- load 可以越过本地 store 流水线中前面的 store, 只要它们的地址不相同;
- load 不能从本地 store 流水线 (无前递功能) 中返回值;
- 存储系统 (本地 store 流水线之后的部分) 满足 store 原子性。

上述存储一致性模型满足 store 原子性, 但是对不同地址的访存可以在 cache 中乱序执行, 这是和顺序一致性的主要区别之一。这种模型中, load 不允许绑定非 GP store 的值, 所有试图利用地理位置的局部性来提前返回值 (在值对应的 store 全局完成之前) 的优化行为都是不允许的:

- 不允许 load 从 store 缓冲中返回值;
- 不允许针对无锁 cache 写操作的优化;
- 当 L1 的数据块发生了失效且还在处理中时, 不允许运行在同一个核上的多个线程之间共享 L1 中的数据;
- 和上面类似, 当存在数据块失效且正在处理中时, 不允许对共享 L2 cache 中的值进行共享访问。

在图 7-3 的顺序处理器架构中, 访存满足 store 原子性, 因此 load 返回的值总是全局完成的。由于 load 被阻塞, load-load 和 load-store 顺序性也就可以自动维护。为了保证 store-store 顺序, store 操作必须在 store 缓冲中以 FIFO 顺序逐个全局完成。

这种模型在 IBM 机器中得到采用, 比如 IBM370 指令集架构。

例 7.14 与顺序一致性的区别 为了说明与顺序一致性的区别, 重新回到例 7.10 (Dekker 算法中的执行片段), 证明该例子符合不支持前递的 store-load 放松模型。

```
INIT: A=B=0
T1      T2
S1(A) 1  S2(B) 1
L1(B) 0  L2(A) 0
```

图 7-29 给出了程序的执行流图, 例 7.10 (图 7-23) 和图 7-29 的区别在于每个线程中 store 和 load 之间的边被删掉了, 因为当 load 访问的是不同地址时, 就可以越过前面的 store。

顺序一致性模型中存在的环在这里已经没有了, 因此所有访问可以按照一致性方式进行排序, 如图 7-29 中所示。该结果在放松的存储一致性模型下是完全合法的, 可见在这种放松模型下, Dekker 算法会失败, 无法实现锁的功能。

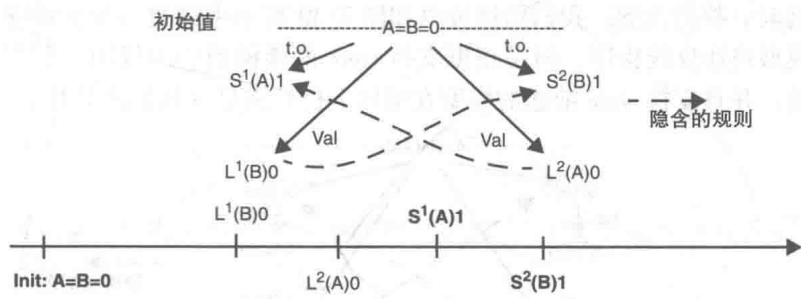


图 7-29 不支持前递的 store-load 放松模型下的程序执行流图

通常情况下，如果某个存储模型对执行顺序的限制更少（即“更弱”），那么合法的执行情况就越多。例如，Dekker 算法的所有执行结果在放松模型下都是正确的，但是在顺序一致性下，有一种执行结果就是错误的。同样，如果程序在某个存储模型中能正确运行，那么在更严格的存储模型中（即“更强”）也一定能正确运行，因为在更强的模型中，可能的执行结果会变少，并且都已经包含在更弱模型的执行结果中了。相反，为某种特定模型编写的程序如果放到一个更弱的模型下运行，结果可能就不对了。比如，为放松内存模型编写的程序总是能够在顺序一致性模型下正确运行，但是为顺序一致性模型编写的程序放到放松模型下，可能运行结果就会有问题（例如 Dekker 算法）。

支持前递机制的 store-to-load 放松模型

支持前递和不支持前递机制的放松模型之间的区别在于，支持前递功能时，load 可以在 store 值还不是 GP 的情况下就从线程的 store 流水线中返回值，因此也违反了 store 原子性，但是整个系统还是满足一致性的。

和前面一样，每个线程的本地 store 流水线之后的存储结构必须满足 store 原子性。这意味着，每当 store 对一个其他线程也可访问的地址进行更新时，这个更新操作就必须是原子的，每个线程的 load 可以从它们各自的 store 流水线中返回值。图 7-27 给出了支持前递的 store-load 放松模型。

因为 store 操作不是原子的，支持前递机制放松模型下的合法执行过程可能无法形成一个包含所有访存操作的全局一致序，而这在顺序一致性和不支持前递的放松模型下都是可以的。因此，其形式化模型也远比以前的模型更复杂。和顺序一致性的做法类似，我们也可以通过一套访存顺序规则来验证执行的有效性。区别在于，由于 load 操作可以直接从本地 store 中返回值并完成，因此，同一个线程内部的 store 和依赖其的 load 之间不进行定序。

支持前递的 store-load 放松模型在 SUN 公司的 Microsystem SPARC ISA 机器上采用，这种模型也常称为全存储序（TSO），下面用一个例子来进行说明。

例 7.15 store-load 放松模型中支持 store 前递的影响 我们用下列代码来突出 store 前递的影响：

```
INIT: A=B=C=0
T1      T2
S1(A)1  S2(B)1
S1(C)1  S2(C)2
L1(C)1  L2(C)2
L1(B)0  L2(A)0
```

支持 store 前递机制时，这两个线程中 store C 和 load C 之间的顺序依赖都被消除了。通过去掉这两个顺序上的依赖，就消除了图中的环。因此当 store 支持前递时，执行有效，而当

store 不支持前递时，执行无效。我们仍然可以如图 7-30 所示来构造一个全局序。由于对 store 前递的支持，模型将违反线程序，但是根据支持 store 前递模型中的规则，所有访存的整体顺序仍然是有效的。并且支持 store 前递的模型在整体上仍然满足（纯）一致性。

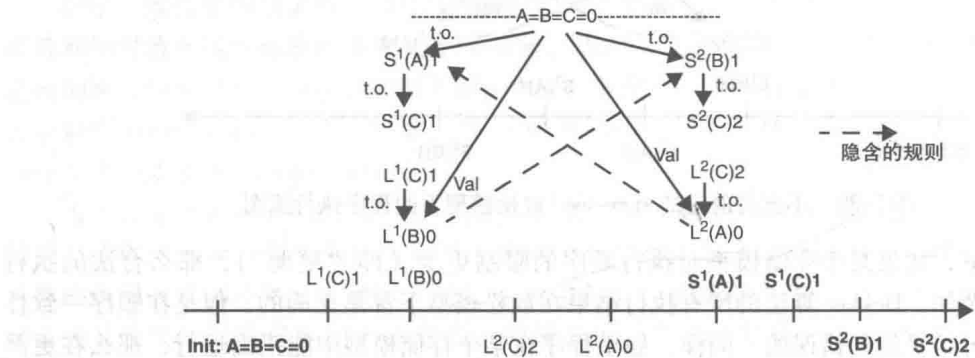


图 7-30 支持前递的 store-load 放松模型下的程序执行流图

这个例子也反映了 7.3.4 节提到的纯一致性问题，即纯一致性不能和其他顺序进行组合。通常来讲，在纯一致性中增加了其他顺序约束（例如 load-load）后，并不总是形成包含所有访存操作的全局一致序。也正是由于这样的全局一致序并不总是存在的，对支持 store 前递模型的证明也比其他模型更加困难。不过，测试程序可以为执行过程构造一个数据流图，迭代地应用模型规则并验证图中不存在环。而形式化的验证方法由于依赖于全局序和人工验证，操作起来就要复杂得多。

大家可能想知道除了放松 store-load 之外，是否还可以通过放松其他顺序性来提高架构性能呢？有时候，保证 load-load 和 load-store 的顺序，是为了确保线程内的依赖关系（地址或值）。比如，load 的地址可能依赖于一个先前的 load，或者 store 的地址/数据可能依赖于先前的某个 load。如果我们获得一个所有访存的全局序，并且其中某个 load 排在相同线程中依赖它的 load 或者 store 之后，那么这将违反线程内的依赖关系。load-load 和 load-store 顺序维护的主要开销在于，没有在 store 缓冲中命中的 load 操作必须在全局完成之后才能返回值。最后一个顺序是 store-store，由于 store 缓冲的引入，store 操作已经不在关键路径上了，唯一的问题是当 store 缓冲满时，可能会导致处理器阻塞。在 Sun 公司的 PSO（Partial Store Order）存储一致性模型（本书中将不对 PSO 进行介绍），以及依赖于同步的存储一致性模型中，允许对 store-store 顺序进行放松。这一放松使得 store 缓冲中的 store 可以乱序并发地传播，因此在某些情况下，可能会有性能的提升，上述情况导致了 store-store 的顺序性被打破。

Sun Microsystem 中的放松存储序

就像所有单处理器一样，在放松存储顺序（Relaxed Memory Order, RMO）中，我们也只确保线程内的一致性。RMO 不会引入任何线程间的顺序约束，不过，指令系统可以通过软件控制的 MEMBAR 指令来确保全局的访存序，同时向程序员和编译器暴露出充分的访存并行性。编译器将 MEMBAR 指令插入到代码中的适当位置，以实现图 7-28 中提到的 4 种顺序性约束：load-load，load-store，store-store 和 store-load。

MEMBAR 指令在功能上类似于线程中内存访问的栅栏，它带有一个 4bit 的操作数，每一位代表下面的 4 种顺序之一：load-load，load-store，store-load 和 store-store。通过在所有的访存操作对之间插入操作数为 1111 的 MEMBAR 指令，编译器可以确保硬件以线程序全局完成所有的内存访问，从而实现顺序一致性，其代价是代码膨胀。RMO 是一个非常灵活的模型，访存操作的顺序性（实际的存储一致性模型）是完全由软件来控制的。

例 7.16 在代码中添加 MEMBAR 指令 在下面的代码中，如何增加 MEMBAR 指令以支持 TSO 模型：

T1	T2	T3
A=1	R1=A	R2=B
	B=1	R3=A

在 T2 中，我们必须保证全局的 load-store 顺序，在 T3 中我们必须保证全局的 load-load 顺序。因此，新的代码如下：

T1	T2	T3
A=1	R1=A	R2=B
	MEMBAR 0100	MEMBAR 1000
	B=1	R3=A

即使 store 向 load 前递值，即使 load 可以乱序执行，即使 cache 以下的存储层次不满足原子性，上面这段代码也只可能产生符合 TSO 的结果（当然，也符合顺序一致性）。T2 中的 MEMBAR 指令确保了在 store 发射到 cache 之前，前面的 load 已经全局完成，T3 中的 MEMBAR 确保了第一条 load 在第二条 load 执行之前就已经全局完成。

需要注意的是，MEMBAR 是一个本地的线程内的访存栅栏，它不同于 barrier 同步，后者是一个全局的、线程间的同步机制，可以影响多个线程的执行顺序。

7.6.2 依赖同步的放松模型

目前为止我们所探讨的所有放松存储模型都是基于硬件实现效率和简化来考虑的（主要是通过放松 store-load 顺序）而与程序员的期望无关。然而，也还存在一些其他的模型，相比于顺序一致性，对硬件的限制会更少，这些模型依赖于同步操作的语义。

无论是否存在多个 cache 副本，也无无论是什么存储模式下，我们都需要同步操作。当共享变量被多个线程读写时，需要通过临界区进行保护，这可以通过锁或 barrier 来实现。在同步点，不同线程彼此交换各自的执行位置信息。通常，同步可以用 RMW 原子指令来实现。如果使用对共享数据的 load 和 store 来同步（例如，Dekker 算法），那么程序员应该知道这一特殊功能，他们需要将特殊共享数据声明为同步变量，这样硬件可以将它们与其他共享变量进行区分，硬件只需要保证同步变量访问时的正确交错顺序，而不需要考虑所有共享变量访问的交错。

为了更好地说明这一点，给出如下的简单程序，通过 FLAG 进行数据同步：

```
INIT: A=FLAG=0
declare SYNC FLAG

T1      T2
...
A=1
FLAG=1  /release
while (FLAG==0) /acquire
print A
```

在这段代码中，A 是普通的共享变量，FLAG 是用于同步的特殊共享变量，并且已声明为同步变量。大部分共享变量都是非同步变量，软件可以传递此信息给硬件，以便硬件只将同步变量看作栅栏（就像顺序一致性中的所有访存操作，或 Sun 的 RMO 中的 MEMBAR 指令）。使用 RMW 指令（比如 T&S 和 swap）访问的共享变量位置都自动被看成是同步变量。

再看下面的例子，其结果的正确性依赖于顺序一致性：

```
INIT: A=B=0
T1          T2
A=1         R1=B
B=2         R2=A
           R3=R1+R2
```

在顺序一致性的情况下，R3 寄存器可能的值是 0，1 或者 3。在没有对访存操作确保全局序的 RMO 系统中，硬件有可能在将 store B 传播到 T2 之后才完成 store A 的传播，因此 R3 得到的值是 2。为了避免这种结果，可以将这部分代码放到临界区内：

```
INIT A=B=0
declare lock L
T1          T2
Lock (L)    Lock (L)
A=1         R1=B
B=2         R2=A
Unlock (L)  Unlock (L)
           R3=R1+R2
```

临界区的存在使得两个线程不能同时执行临界区内代码。因此，T1 和 T2 对 A 和 B 的访问将永远不能并发执行，所有可能的执行都将满足顺序一致性。事实上，即使 store 不具有原子性，临界区内指令的执行效果看起来也是原子的。唯一的可能结果是 R3 = 0 或 3，这两个结果都符合顺序一致性。

为了确保上述代码的正确执行，硬件必须区分对锁变量 L 的访问以及对 A 和 B 的访问。线程 T1 中，并不要求在执行 store B 之前一定要全局完成 store A，然而，当执行到访问 L 时，之前所有的访存操作必须都已经全局完成。

放松存储一致性模型将同步变量访问看作栅栏操作，而对其他变量的访问没有顺序要求，这种模型也被为弱顺序模型。（相对于强顺序模型而言，强顺序模型会在硬件上对所有访存操作进行排序。）

弱顺序模型（弱一致性模型）

线程包含临界区和非临界区代码段，在非临界区代码段中，只可以访问私有变量或只读共享变量。在临界区代码段中，所有共享位置互斥访问。因此，对相同地址的 store 操作可以通过由 lock 和 unlock 构成的临界区来进行排序。

图 7-31 是由 3 个线程构成的一个程序片段。该程序有两个关键部分：一个是对变量 X 访问的保护，另一个是对变量 B 访问的保护，其中变量 B 是一个 barrier 计数器。X 和 B 都是普通共享变量，更新操作受到临界区的保护。读取 B 的 while 循环没有必要放在临界区内，因为 B 是单调增加的，并且每个线程只有在 B 达到最大值时才能读取到 B。图中的箭头显示了弱顺序模型中引入的全局序。

在弱顺序一致性模型中，线程中的 RMW 指令（lock 操作）和 swap 指令（或带标记的 store 指令，unlock 操作）作为每个线程代码操作中的栅栏。unlock 操作不能像常规 store 一样实现，需要能够被硬件识别出。一些其他共享变量也可以声明为同步变量，比如 FLAG 可以用于变量通信的同步，或者是 Dekker 算法中的标志同步。对这些同步变量的访问统称为同步访问。对于参数为 1111 的 MEMBAR 指令，所有之前的访存操作必须在同步访问发射之前就全局完成，并且在同步操作全局完成之前，后续的访存操作不能发射到内存。因为 RMW 指令同时包含 load 和 store，我们需要对“全局完成”的定义进行补充，以便适用于 RMW 访问。

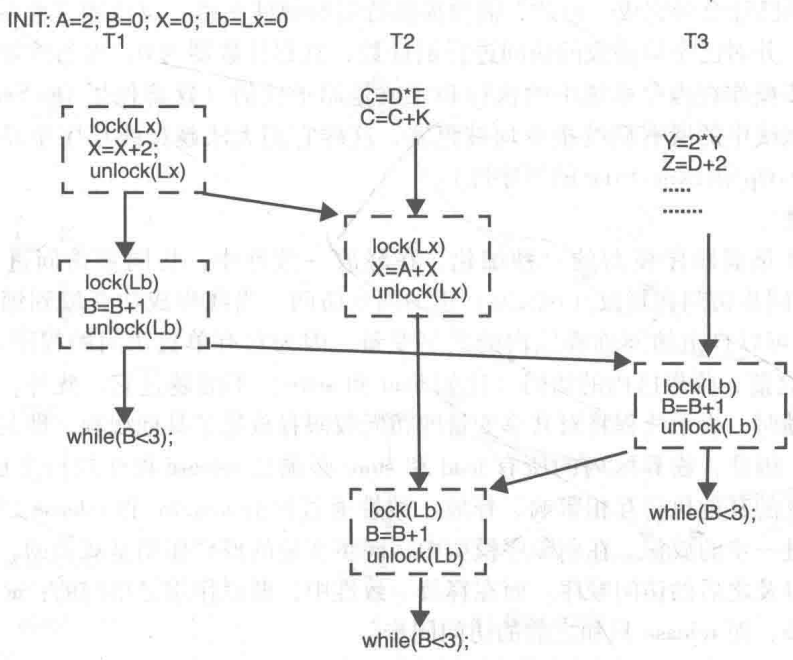


图 7-31 支持同步的多线程程序示例

定义 7.12 (RMW 访问全局完成) 当 RMW 访问中的 load 和 store 都已经全局完成时，我们称对某个内存位置的 RMW 访问已经全局完成。

RMW 访问必须满足原子性，也就是说，在 RMW 的 load 和 store 之间，不允许有其他线程对该内存位置进行更新。基于无效策略的 cache 协议更有助于保证 RMW 的原子性，如果 cache 协议把 RMW 访问看成一个 store 操作，那么必须先获得数据块的一个已修改的副本（唯一副本），然后 load 和 store 可以在 cache 中原子执行，因为没有其他线程可以访问到该副本，在 RMW 访问完成之前，其他线程的访问请求都会被拒绝。

在对全局完成的定义中，全局 store 顺序不一定要通过 store 原子性或者一致性来实现，而是借助临界区和同步访问来完成。对可修改共享变量的访问代码可以通过同步访问进行隔离。在两次同步访问之间，访存操作可以按照任意顺序发射和完成，只要能够满足线程内的访存依赖关系即可。

图 7-32 给出了弱顺序模型下确保的顺序性，图中的“Op”是 load 或 store，“Sync”是指对任何同步变量的访问，包括锁，在普通的 load 和 store 之间没有顺序要求。

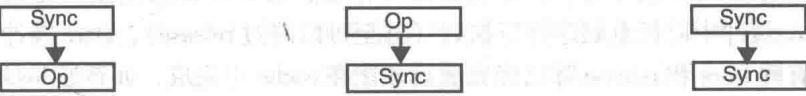


图 7-32 弱顺序模型下的序要求

在图 7-3 的顺序微架构中，所有指令（包括同步）都按照线程程序执行。在弱顺序模型系统中，store 被插入到 store 缓冲中，load 可以在 store 完成之前越过其执行，并且可以用 store 缓冲中的值进行前递。上述优化已经在 TSO 模型中采用，弱顺序模型的另一个优化方法是，store 缓冲中的普通 store 操作可以按任意顺序并行执行（无 store-store 的顺序要求）。

对同步变量（不管是 load、store 或 RMW）的访问则必须区别对待。它们必须在流水线的访存阶段中等待，直到 store buffer 中的所有 store 都已经全局完成。此外，同步操作之前的所有

load 操作也必须已经全局完成。因此, 需要提供特定的硬件支持, 以检测之前的所有访问是否已经全局完成, 并对已全局完成的访问进行倒计时, 直到计数器为 0, 然后才能执行对同步变量的访问。同步操作在内存系统中的执行和完成是原子性的 (这确保了 Op-Sync 的顺序性)。与此同时, 流水线中的所有后续指令均被阻塞, 这样它们无法越过线程序中靠前的同步访问 (这确保了 Sync-Op 和 Sync-Sync 的顺序性)。

释放一致性

释放一致性是弱顺序模型的一种细化, 在释放一致性中, 将同步访问进一步分为获取 (acquire) 锁的同步访问和释放 (release) 锁的同步访问。当线程成功获取到锁时 (类似于打开临界区), 就可以自由访问临界区内的共享变量, 因为它有单独访问的权限。因此, 在 acquire 全局完成之前, 临界区内的访问 (比如 load 和 store) 不能越过它。此外, 当线程释放了当前临界区的锁时, 表示线程将对共享变量的访问权限释放给了其他线程, 即其他线程将有权修改共享变量。因此, 临界区内的所有 load 和 store 必须在 release 操作执行之前就全局完成, 以避免和临界区的其他执行互相影响。释放一致性通过区分 acquire 和 release 之间的不同对弱顺序模型做了进一步的放松。在弱顺序模型中, 同步变量的栅栏作用是双向的, 即同时限制了同步操作之前以及之后的访问顺序。而在释放一致性中, 栅栏作用是单向的: acquire 只和之后的访问进行同步, 而 release 只和之前的访问同步。

释放一致性对顺序性的要求如图 7-33 所示, 普通的 load 和 store 之间没有顺序性要求。和弱顺序模型一样, 所有的同步访问必须在线程间满足顺序一致性。这也是出于编程中临界区模型的要求。

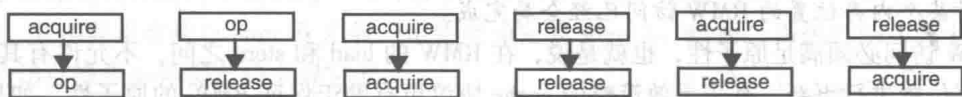


图 7-33 释放一致性中的顺序性

相比弱顺序模型, 释放一致性可以暴露更多的访存并发性, 还以图 7-31 中线程 T2 的代码为例, 图 7-34 给出了两者在访存并行性方面的区别。需要注意的是, 在释放一致性中, acquire (lock) 以及临界区内的操作, 可以和 acquire 之前的代码同时执行 (假设不存在依赖关系), 同样, 在释放一致性中, 临界区内的代码以及 release (unlock) 操作, 也可以和 release 之后的代码同时执行 (也假定不存在依赖关系)。这些潜在的并行性在弱顺序模型中是没有的。

在图 7-3 的顺序微架构中, 这种额外的并行性主要意味着 release 操作可以按照 store 顺序缓存在 store 缓冲中, 因此, 当临界区内的 store 还在执行时, 可以允许后续的 load 和 store 先完成。release 在发送到 store 缓冲之前, 必须确保所有之前的 load 都已经全局完成。而普通 store 操作可以在 store 缓冲中以任意顺序并行执行 (甚至可以跨过 release)。store 缓冲中的 release 需要等到之前所有的 store 和 release 都已经完成后才能在 cache 中完成, 而普通 load 操作不需要等待 store 缓冲中的普通 store 或 release, 可以直接进行值的前递。

acquire 操作必须先访存流水级中全局完成, 后续的其他访存操作才能完成 (确保 acquire-op, acquire-acquire, acquire-release 的顺序性)。这在硬件上是可以自动保证的, 因为 acquire 会阻塞流水线, 直到全局完成。此外, acquire 必须等待 store 缓冲中的所有 release 先全局完成 (release-acquire 顺序性要求), 但是不需要等待普通 store 操作。

虽然释放一致性做了进一步放松, 相比于弱顺序模型在硬件层面具有更高的效率, 但是释放一致性要求在编程时对 acquire 和 release 这些同步变量的访问进行标记, 而在弱顺序模型中, 只要求对所有的同步变量进行声明即可。

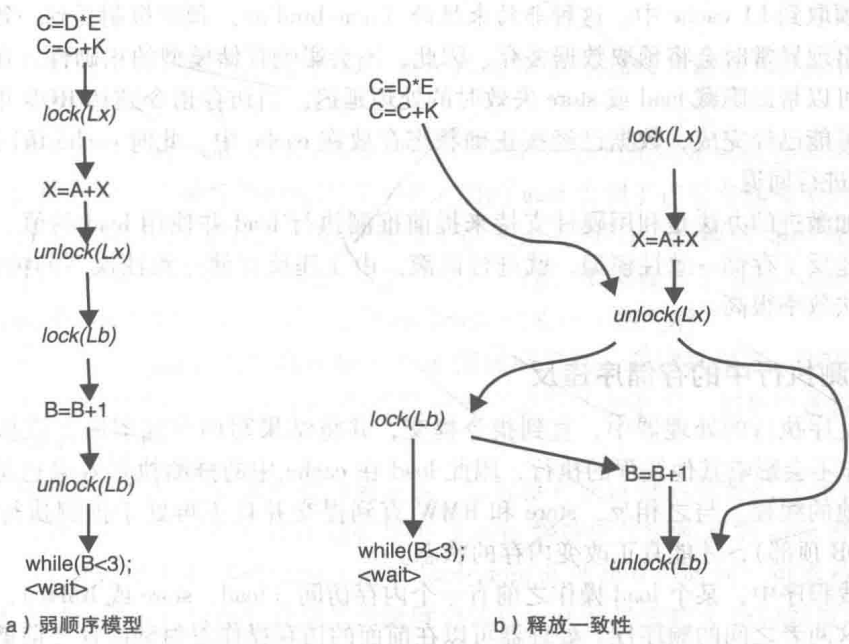


图 7-34 线程 T2 执行过程中的并行性

7.7 推测执行中的存储序违反

目前为止，我们主要考虑的是顺序流水线处理器结构（例如图 7-3 中的流水线），然而在最新的处理器中，load 通常可以推测乱序执行。在支持推测乱序执行的处理器中，可能同时有几十、甚至上百条指令在执行。指令按照线程序进行取指、译码、分发，然后，一旦输入操作数准备好，就立即以数据流的顺序开始执行。当执行结束时，又重新按照线程序在重排序缓存（ROB）中等待，直到轮到它们更新寄存器或者内存时，再将指令退出。在指令分发和退出之间，指令推测执行，当发现之前的分支指令预测错误，或者之前指令出现异常时可以进行回滚。支持推测乱序执行的处理器具有回滚所需的硬件结构，通常情况下，我们使用这些硬件结构来预测某些特定事件会不会发生，然后指令根据预测结果进行推测执行，当检测到预测错误时，就对执行过程进行回滚。显然，只有当预测准确性非常高时，上述策略才能提高性能，因为回滚执行会造成很大的开销。

7.7.1 乱序执行处理器中的保守存储模型

乍看起来，在推测的乱序执行中，似乎很难对访存操作进行任何形式的定序，实际上，在推测乱序执行的处理器中，load 和 store 操作只有在准备好提交和退出时才算是全局完成。尽管 load 可以在指令退出前先返回推测值，并且可能被后续指令继续使用，但是一旦检测到冲突（即之前的推测是错误的），返回的推测值还可以被撤回，因此 load 要一直到达 ROB 顶部，并且值不可撤回时，load 值才算真正被锁定了。而对于 store 和同步指令，它们也只有在到达 ROB 顶部时才能更新内存，因此，也需要等到准备好退出时才算全局完成。

确保符合存储模型的一种保守方法是等到 load 到达 ROB 顶部时才允许返回值。不过，这种方法会严重影响处理器的性能，导致处理器的访存开销很大，并且 load 操作之间的指令并行度非常有限。

要提高这种保守方案的性能，处理器可以在 load 和 store 的地址已知的时候，就立刻将对

应的内存块预取到 L1 cache 中。这种非约束性的 (non-binding) 预取机制受到一致性协议的限制, 并且在出现异常时会将预取数据丢弃, 因此, 不会影响存储模型的正确性。在效果上, 至少这种预取可以帮助隐藏 load 或 store 失效时的处理延迟。当访存指令到达 ROB 顶部时, 数据块预取操作可能已经完成, 数据已经按正确状态存放在 cache 中, 此时 cache 访问命中并且不需要对 ROB 进行回退。

一种更加激进的办法是利用硬件支持来提前推测执行 load 并使用 load 的值, 如果检测到该推测执行违反了存储一致性模型, 就进行回滚。由于违反存储一致性模型的情况很少出现, 因此这种方法效率很高。

7.7.2 推测执行中的存储序违反

在推测乱序执行的处理器中, 直到指令提交, 并将结果写回存储器时, 该指令才真正生效。load 操作不会影响其他线程的执行, 因此 load 在 cache 中的推测执行效果也是本地的, 不会影响到其他的线程。与之相反, store 和 RMW 直到提交并且不再处于推测执行状态时 (比如, 到达 ROB 顶部), 才能真正改变内存的状态。

假定在线程序中, 某个 load 操作之前有一个内存访问 (load、store 或 RMW), 并且存储模型要求确保这两者之间的顺序性。处理器可以在前面的访存操作发射到内存之前就先发射, 并且推测执行后面的 load 操作, load 返回的推测值可以被后续的推测指令继续使用。此后, 当 load 准备退出, 并且之前所有的访存操作都已经退出时, load 返回的推测值需要和重新访问 cache 获取到的当前值进行比较, 如果值不相同, 那么必须回滚到 load 处, 如果值相同, 那么针对 load 的推测是有效的, 并且 load 在 cache 副本中全局完成。这个结果和 load 保守执行时 (load 到达 ROB 顶部再处理) 的结果是一致的。这种方法的问题在于, 每次 load 操作都需要对 cache 访问两次。

另一种用来检测违反存储序的安全高效的解决方案是借助 cache 和 cache 一致性协议。load 操作尽可能快地推测执行并返回值, 同时, 包含该内存位置的数据块被读入处理器节点的 cache 层次结构中 (假定之前不在 cache 中)。如果有某个远程处理器试图修改 load 返回来的数据, 就会往该处理器节点发送一个通知 (更新或无效) 消息。该通知消息可以通过监听 load/store 队列来检查 load 操作是否已经推测完成。如果已完成, load 和所有后续的推测指令的执行都必须进行回滚 (类似分支预测错误), 因为推测执行的过程可能会违反存储模型, 并且 load 返回的值可能已经影响到了后续的指令。当 load 已经被推测执行, 并且在没有撤回的情况下到达 ROB 顶部, 那么该 load 操作不需要访问 cache 就可以全局完成, 然后退出。在这个机制下, 在 load 推测执行的整个过程中, 包含 load 值的数据块必须始终在本地 cache 层次中, 以便仍然遵守一致性协议。在这段时间内, 如果 cache 必须将该数据块替换出节点, 那么 cache 结构必须触发一个违规信号。当然, 上述解决方案在不含 cache 的系统或者含有 cache 但是没有或者只有部分硬件一致性支持的系统中无法工作。

要判断推测执行是否违反存储模型约定, 一种方法是分别记录 load 推测执行的时间以及 load 操作退出的时间。如果在这两个时间点之间没有出现改变 load 值的远程事件, 则后续依赖于 load 值的指令所获得的操作数就是正确的, 执行过程也是有效的, 因此 load 操作可以安全退出。相反, 如果两个时间点之间发生了改变 load 值的远程事件, 那么 load 的值必须被撤回, load 后面的指令也必须进行回滚, 这一机制称为 load 值撤回 (load value recall)。接下来, 我们利用这种机制来保证带推测执行的乱序结构中的存储一致性。

顺序一致性中的推测执行

在推测顺序一致性中, 图 7-28a 中的所有顺序约束必须进行全局维护, 下面我们逐个

说明。

load-store 顺序可以自动维护, 因为 store 必须到达 ROB 顶部时才能在 cache 中完成, 这之前所有的 load 都已经全局完成并且退出。

load-load 顺序可以通过 load 值撤回来维护。load 操作一旦地址确定后就会推测执行, 如果 load 在提交之前被撤回, 表明可能相对线程中之前的 load 出现了违反存储模型的情况。

store 以线程在 ROB 顶部退出, 并插入到先进先出 (FIFO) 结构的 store 缓冲中, 然后 store 操作按照 FIFO 顺序在 store 缓冲中逐个全局完成。这一策略保证了 store-store 顺序, 因为 store 操作需要确保之前所有的 store 都按照线程全局完成后才能发射。

为了确保 store-load 顺序, 我们再次借助 load 值撤回机制。在该机制下, ROB 顶部的 load 操作要等到 store 缓冲中之前所有的 store 都已经全局完成后才能退出。如果可能的话, load 也可以在 store 缓冲中返回值。

虽然推测乱序执行允许 load 被推测全局完成, 以提供对 load 延迟的容忍 (在 cache 失效的情况下), 但是到达 ROB 顶部的 load 还是必须等待 store 缓冲中之前所有的 store 都按照线程逐个全局完成。此外, ROB 和其他结构 (诸如 load/store 队列和发射队列) 可能被填满, 一旦 ROB 或其他队列之一被填满, 那么分发流水级就会堵住, 从而导致整个处理器被阻塞。因此, 即使在支持乱序执行和 load 值撤回的情况下, 放松模型通过允许 load 越过之前 store, 仍然可以获得比顺序一致性模型更高的效率。

TSO 中的推测执行

在 TSO 模型中, load 可以从线程的 store 流水线中返回一个非 GP 值, 以此在本地推测完成, 虽然这个返回的值仍然在线程的 store 缓冲中, 但是推测完成的 load 却不能撤回, 因为 load 对应的位置不一定在 cache 中有缓存。但是这仍然是和模型一致的, 因为 TSO 允许线程从 store 流水线中返回非 GP 值。

如果当推测 load 退出的时候, 返回其值的 store 仍在 store 缓冲中, 那么此时是和模型一致的。与之相反, 如果 store 在 load 退出之前就已经全局完成, 那么数据块副本将会加载到 cache 中, load 也将受到无效或者更新信号的约束, 这也与模型一致。

在支持 store-load 顺序放松的模型中, 乱序处理器的 store 缓冲是有作用的。不过, store 缓冲中的 store 仍然必须以线程全局完成。弱顺序模型和释放一致性中没有这些限制, 普通 load 和 store 操作可以在存储系统中乱序执行。下面我们先介绍一下在推测乱序执行处理器中, RMW 访问是如何推测执行的。

RMW 访问中的推测执行

RMW 访问包含一系列复杂指令: 先是一个 load, 然后是一些寄存器操作, 最后是一个 store, 整个操作序列必须原子完成。

先看一个最简单的例子: test_and_set, 这条指令先是一个 load, 紧接着一个 store 1 的操作。如果支持 load 值撤回, 那么 test_and_set 可以当成 load 推测执行, 当 test_and_set 到达 ROB 顶部时, 又被看成一个 store 操作来对其全局完成。因为 test_and_set 首先被看成一条 load, 如果在退出之前收到更新或者无效请求消息, 则 load 值将被撤回。如果 test_and_set 在退出之前被撤回, 那么后续的整个推测执行过程都将进行回滚。该 test_and_set 需要等到之前所有的访存都已经全局完成时, 它才能全局完成并退出。因此, test_and_set 必须在 ROB 顶部等待, 直到 store 缓冲中所有之前的 store 操作都已经全局完成, 其对应的 load 操作仍然支持 load 值撤回机制来保障。最后, test_and_set 不允许在 store 缓冲项上进行推测执行 (即不对 test_and_set 中的 load 进行前递)。

弱顺序模型中的推测执行

在弱顺序模型中，对同步变量的所有访问（同步 load，同步 store 或 RMW 指令）都将视为栅栏操作（类似顺序一致性中的 load 或者 store 或者 RMO 中的 MEMBAR 指令）。对同步变量的访问必须到达 ROB 顶部（不再处于推测状态），并且确保 store 缓冲中的所有 store 都已经全局完成时，它才能全局完成，此外，所有之前的 load 操作也必须都已经全局完成。上述条件确保在执行同步变量访问之前，之前所有的访存操作都已经全局完成了。

我们还需要确保只有当同步变量访问已经全局完成后，后续的访存操作才能执行。这一点对于 store 和 RMW 访问来讲是自动保证的，因为 store 不允许推测执行。当有 load 值撤回机制保障时，对同步变量的 load 访问（包括 RMW 访问中的 load）可以推测执行，而 RMW 访问中的 store 操作必须等待 store 缓冲为空并且之前所有的 load 都已经全局完成后，才能在 ROB 顶部完成。上述优化使得对同步变量的 load 操作或 RMW 访问操作之后的其他指令可以推测执行。对非同步变量的普通 load 操作也可以推测执行，但是无需受限于 load 值撤回机制，如果之前的同步操作或者 RMW 访问被撤回，那么该推测执行过程也会进行回滚。

释放一致性的推测执行

在释放一致性中，release 操作要等到之前所有的访存操作都已经全局完成之后才能执行，而 acquire 后的访存操作也需要等到 acquire 已经全局完成之后才能执行。不过，load 操作可以推测执行。

和弱顺序模型中类似，所有对非同步变量的普通 load 操作可以在到达 ROB 顶部之前就推测执行，无需受 load 值撤回机制的限制。acquire 中的 load 操作也可以推测执行，但是必须有 load 值撤回机制的保障，这样当出现违反模型定义的情况时可以对推测执行过程进行回滚。根据释放一致性的定义，acquire 可以越过之前的 store，但是不能越过之前的 release。在 store 缓冲中的所有 release 全局完成之前，acquire 必须在 ROB 顶部等待，以便保证可以撤回。

在乱序处理器中，释放一致性相比弱顺序模型的好处，和图 7-3 中顺序处理器中的情况是一样的：作为 release 操作中的一部分同步访问（假定是带标记的 store，而不是 RMW 指令）可以像普通 store 一样插入到 store 缓冲中，从而使得后续的其他指令可以先退出。

习题

7.1 (a) 考虑如下的程序段，假设 A 和 B 是内存中变量，并且初始值为 0，R1、R2 和 R3 是寄存器：

P1	P2	P3
A=1	R1=A	R2=B
	B=1	R3=A

程序的不同执行过程可以通过最后 load 的返回值来区分，即执行结束时 R1、R2 和 R3 中的值。分别给出下列情况下的执行过程和寄存器值（如果存在的话）：

- (a1) 不满足纯一致性。
- (a2) 不满足顺序一致性。
- (a3) 不满足 TSO。
- (a4) 不满足弱顺序模型。

针对上面每种情况给出你的理由。

(b) 针对下面的程序，回答 (a) 中的同样问题：

P1	P2
A=1	B=1
R1=B	R2=A

(c) 针对下面的程序，回答 (a) 中的同样问题：

P1
A=1
R1=A
R2=B

P2
B=1
R3=B
R4=:A

(d) 针对下面的程序，回答 (a) 中的同样问题：

P1
A=1
C=1
R1=C
R2=B

P2
B=1
C=2
R3=C
R4=A

7.2 本题中，使用不同的同步原语来实现简单的 barrier 同步代码，以支持 N 个线程的同步：

BARDEC BAR=0

P1	P2	PN
...
BAR +=BAR;	BAR +=BAR;	BAR +=BAR;	
while(BAR<N);	while(BAR<N);	while(BAR<N);	
...	

很显然，增加变量 BAR 值 (barrier 计数器) 的语句必须使用某种同步原语进行保护，否则某些增加 BAR 值的操作可能没有生效，从而导致代码出现死锁。

- (a) 请解释为什么不需要用临界区来保护 while 循环中对 BAR 变量的读取。
- (b) 使用 test_and_test&set 指令，给出每个线程的正确实现代码 (用类 MIPS 汇编代码给出所有指令)。假设有一条新的 T&S R1, X 指令，其中 X 是内存地址。
- (c) 假定有一条新的 F&A 指令 (F&A R1, X)，F&A 包含一个加 1 操作，X 是内存地址。给出每个线程的正确代码实现。
- (d) 上述 barrier 代码的问题之一是 barrier 不允许多次进入，因为在 barrier 同步的最后，BAR 的值是 N ，为此给出下面的代码 (我们还是假定 BAR 增加的指令是放在临界区内的)：

BARDEC BAR=0

P1	P2PN
...
BAR +=BAR;	BAR +=BAR;	BAR +=BAR;
if (BAR==N) BAR=0;	if (BAR==N) BAR=0;	if (BAR==N) BAR=0;
while(BAR!=0);	while(BAR!=0);	while(BAR!=0);
...
BAR +=BAR;	BAR +=BAR;	BAR +=BAR;
if (BAR==N) BAR=0;	if (BAR==N) BAR=0;	if (BAR==N) BAR=0;
while(BAR!=0);	while(BAR!=0);	while(BAR!=0);
...

我们可以通过使用 CAS (R1, R2, X) 指令来原子执行上面的 if 语句，请解释如何实现。此外，我们并没有必要原子执行 if 语句，请解释为什么。

上述代码可能导致死锁，请解释为什么。

- (e) 问题 (d) 的解决方案之一是交替地增加和减少 barrier 计数。我们只需要记录一个二进制标志用于表示 barrier 已经执行了偶数次 (增加 BAR) 还是奇数次 (减小 BAR)。请使用 F&A 指令实现一个 BARRIER (BAR, N) 函数，要求函数可以任意多次调用，并且每次均可正确工作。

7.3 本题考虑用 LL (load-linked) 和 SC (store conditional) 实现各种同步原语。两种基本指令格式如下：

LL Rx,A/load mem location A in Rx;

SC Rx,A/conditionally store (Rx) into mem location A

store 是有条件的，如果处理器在 LL 和 SC 之间收到针对 X 的无效或者更新请求消息，那么 store 将被终止，Rx 返回 0。（注意，当 lock 成功时 store 的值不可能是 0）。store 成功与否可以通过侦听器或者网络接口中的一些寄存器来检测，寄存器中保存了 LL 的地址，并且每当 SC 执行时进行检查，如果网络接口收到修改该地址的无效或者更新请求消息，SC 将会失败。

在 LL 和 SC 之间可以插入其他指令，只要插入的指令不是 store 操作，并且不会触发异常。比如，指令本身可能会产生意外，但是程序员知道在这种情况下异常永远不会出现，那么也是可以使用的。在 LL 和 SC 之间进行上下文切换会导致 SC 失败。

在本章中，我们已经介绍了如何使用上述指令实现一个 test_and_set 功能，这里，将使用 LL 和 SC 实现一些其他的同步原语（甚至有些实际不存在的同步原语）：

- (a) 给出 F&ADD X, Rx, a 的实现代码，其中 a 是加到 X 上的一个小的立即数。
- (b) 给出 F&ADD X, Rx, Ry 的实现代码，其中 Ry 加到 X 中，并且 Rx 返回 X 在执行 ADD 之前的值。
- (c) 给出 F&ADD X, Rx, Y 的实现代码，其中内存位置 Y 的值加到内存位置 X 中，执行 ADD 前的 X 值存入 Rx。
- (d) 给出 SWAP (Rx, X) 的实现代码，其中 Rx 和 X 的值将进行交换。
- (e) 给出 CAS Rx, Ry, X 的实现代码，先将 Rx 的值与 X 的值进行比较，如果相等，则将 Ry 和 X 的值进行交换。
- (f) 要求程序员使用 LL 和 SC，而不是使用锁，来实现一个像原子操作的临界区，线程的临界区代码如下：

```
A += A;  
if (A==8) B=B+1;  
else B=0;
```

A 和 B 是可写的共享变量。假定程序员直接用汇编语言来编写代码，可否找到一种使用 LL 和 SC 来实现临界区代码的方法？如果有，请给出代码；如果不能，请解释原因。

7.4 为了提高效率，多处理器系统中的每个处理器都必须配备 store 缓冲，就像单处理器一样，本题将探讨多核环境下 store 缓冲的管理问题。

假设处理器每次只执行一条指令（即没有流水线）。参考图 7-35，假定处理器没有任何 cache，并通过电路交换总线连接到内存中，内存由单个内存 bank 构成（因此 cache 访问是原子的）。在任何情况下，store 一旦插入到 store 缓冲中就可以完成（退出，提交），此后 store 缓冲逐个处理针对内存的 store 操作。

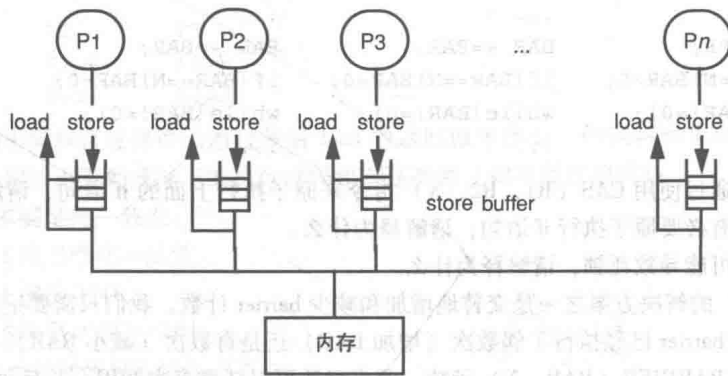


图 7-35

大多数 load 都跳过 store 缓冲直接访问内存，当值返回时 load 操作也就完成了。不过有些 load 可能从 store 缓冲中返回值（参考下面的说明）。在下面给出的每种情况中，分别说明多处理器是否满足纯一致性、顺序一致性、store 原子性、弱顺序模型或者 TSO，并说明原因。

- (a) 当 store 还在 store 缓冲中时, store 后面相同地址的 load 操作可以通过返回 store 缓冲中与该 store 相关联的值而完成 load 操作。store 缓冲按照 FIFO 方式管理, 如果在 store 缓冲中有一个以上的对同一地址的 store, 那么 load 以 FIFO 顺序返回最近的 store 值。store 操作在 store 缓冲中按照 FIFO 顺序逐个全局完成, 当 store 操作还在存储系统中传播时, 其对应的项一直保存在 store 缓冲中。
- (b) 同 (a), 但是任何时间对于给定地址只能有一个 store 在 store 缓冲中。如果 store 发射到 store 缓冲中时, 里面已经有对应同一地址的 store 了, 那么新的 store 将以 FIFO 顺序替换掉旧的 store。
- (c) 同 (a), 但是当 store 还在 FIFO 结构的 store 缓冲中时, 如果 load 从内存而不是从 store 缓冲中返回值, 那么 store 后相同地址的 load 可以完成。
- (d) 同 (a), 但是当 load 在 store 缓冲中找到有对应相同地址的 store 时, load 需要等待直到 store 在内存中执行, 然后再从内存中读取到相应的值, 从而完成 load 操作。
- (e) 同 (a), 但是所有 load 都需要等到 store 缓冲为空后才能在内存中执行。
- (f) 按照如下条件重复 (a) ~ (e): store 缓冲中的 store 可以在内存中以任何顺序尽可能快地执行, 无需考虑特定的处理顺序。

7.5 侦听 cache 协议可以是基于写无效策略或者写更新策略的, 这两者各有自己的缺点, 一个折中的办法是采用混合的写更新/写无效协议。

在这个折中的协议 (称为竞争协议) 实现中, 基本协议是基于更新的, 然而, 如果 cache 副本在本地处理器访问之前就被远程处理器更新了不止一次, 那么该本地副本将自己无效掉。要实现这一点, 只需要为每个 cache 块设置一个相关位, 称为 UP 位。数据块刚装载进 cache 时, UP 位置 0, 如果接收到更新请求, UP 位置 1, 当处理器进行了本地访问时, UP 位又被重置为 0。如果 cache 块收到更新请求时, UP 位为 1, 那么该数据块就会在本地无效掉。

该协议是一种四状态、写穿透的协议。总线共享线路上刚返回的值在状态图中用 S 表示。S0 是当 UP 为 0 时的共享状态, S1 是 UP 为 1 时的共享状态。在状态 S0 和 S1 时, 数据是干净 (由于采用写穿透策略, 内存是一致的) 和共享的 (可能存在多个副本)。最后一个状态 D 表示数据被修改, 并且是系统中唯一的副本 (内存数据是旧的)。

请推导出这个协议的状态转换图, 对于每一个状态转换, 请给出对应的动作, 类似第 7.3.2 节中介绍的 MSI 协议。有限状态机 (FSM) 的输入是 PrRd、PrWr、BusRd 和 BusUpd。需注意的是, 部分总线访问需要清空数据块, 某些转换过程可能会进入两种不同的状态, 具体取决于总线共享线路返回的值。

7.6 本题探讨 cache 失效对于实际时序和存储一致性模型的敏感度, 我们使用雅可比算法对各种不同 cache 一致性协议下的开销进行说明, 该算法的流程图如图 7-2 所示。

假设 cache 无限大 (即不存在容量失效和冲突失效), 并且算法已经运行了一段时间 (即不存在冷失效)。不过由于 cache 一致性协议的原因, cache 仍然可能存在一致性失效, 例如, 基于无效协议中无效请求导致的失效或者基于更新协议中更新请求导致的失效。由于矩阵 A 是只读的, 对矩阵 A 的访问不会出现失效。此外, 也忽略掉由于 barrier 同步导致的失效。因此, 这里只关注对 X 和 Y 的访问。所有的 X 存在于同一个 cache 块内, 所有的 Y 也存在于同一个 cache 块内。由于 X 和 Y 是可写的共享变量, 因此将其声明为 volatile 类型, 这样编译器不会把它们放到寄存器中。

线程 1 (T1) 中对 Y 和 X 的访问顺序如下:

rY1, wX1, rY2, wX1, rY3, wX1, rY4, wX1, rX1, wY1, rX2, wY1, rX3, wY1, rX4, wY1, ...

其中, r 代表“读”, w 代表“写”。线程 T2、T3 和 T4 对 X 和 Y 访问的顺序也类似。

考虑如下四种协议: MSI-无效, MSI-更新, MESI 协议, 习题 7.5 中的竞争协议。

- (a) 首先假定系统是顺序一致性的, 所有处理器的运行速度完全相同, 并且访存操作必须逐个全

局完成, 这样 4 个线程按照轮询的方式对 X 和 Y 进行交错访问。以上面的执行序列为例, 所有处理器每次都先执行读 Y1, 然后所有处理器再执行写 X, 以此类推。

在 Jacobi 算法整个循环的一次迭代中, 在 MSI、MESI 无效协议下, 所有处理器总的一致性失效次数有多少? 在 MSI-更新协议中, 有多少次更新? 在题 7.5 的竞争协议中, 更新和无效的次数又分别有多少?

(b) 在不同的假定时序下重新回答问题 (a)。仍然假定系统是顺序一致性的, 并且访存操作逐个全局完成。然而, 这里假定访存的全局交错方式有所不同: 在迭代的第一个和第二个阶段, 线程轮流执行访存操作, 先是 T1, 然后 T2, 再 T3, 最后是 T4。然后到第二阶段后, 还是这样的顺序, 从 T1、T2、T3, 最后到 T4。

(c) 假定系统满足释放一致性, 并且 store 缓冲足够大, 因此, store 操作只有在必要时 (比如 release 时) 才会进行值的传播, 针对上面四种协议, 更新失效/无效失效的次数分别是多少?

7.7 考虑如下的访存序列:

r1, w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3 r2 w2 w2 r2

上面的所有访问都是针对相同内存位置的: r 和 w 分别表示读和写, 数字表示是哪个处理器。对比四种协议: MSI-无效, MSI-更新, MESI 协议和竞争协议。在 MSI 和 MESI 中, 新引入一个 BusUpgrade 事务, 该事务在处理器已经有一个共享 (一致且最新的) 副本时使用, 这样我们只需要简单无效掉其他 cache, 而无需发出 BusRdX 并重新装载对应数据块。

假设所有 cache 初始为空, 使用如下的成本模型: 无需总线访问的读/写 cache 命中, 开销为 1 个周期; 需要简单总线事务 (BusUpgrade, BusUpdate) 的 cache 访问, 开销为 20 个周期; 需要传送整个 cache 块的失效开销为 150 个周期。上面的开销数据是包含执行过程中所有访问的总周期数。

在基于总线的机器上执行上述访存序列, 对比在四种不同协议时的开销并解释差异的原因, 给出执行过程中具体的访问和协议流。

7.8 要将 13 个任务 $T(i)$, $i=0, \dots, 12$ 调度到一个包含 4 个处理器的多处理器系统中, 假定任务按照索引号顺序创建 (0~12)。考虑如下四种可能的调度策略:

- 静态调度。按照模 4 的方式静态将任务 $T(i)$ 分配给对应处理器 ($i \bmod 4$)。
- 半静态调度。任务最初采用静态分配, 不过, 当某个处理器空闲时, 就从其他的任务队列中还有待执行任务的处理器中“偷”一个索引号最小的任务进行执行。
- 动态调度。将任务的描述符按照创建的顺序保存在一个 FIFO 工作队列中, 然后每个处理器从队列中获取描述符并执行对应任务, 任务执行完后再重新取下一个描述符, 直到所有任务都已经执行完。
- 最优调度。假定我们一开始就知道任务的执行时间, 因此可以将任务按照最优的方式调度到四个处理器上, 使得总执行时间 (不考虑其他开销) 最短。

请根据如下两种情况分别给出四种调度策略下的总执行时间和加速比: (1) 任务 $T(i)$ 的执行时间为 $1+i$; (2) 任务 $T(i)$ 的执行时间为 $13-i$ 。

7.9 简单的 CAS 指令中有三个操作数: 寄存器 Rx、Ry 和地址 X, 如果 X 的值和 Rx 相等, 那么将 Ry 和 X 的值进行交换。

CAS 指令的描述如下:

```
CAS(Rx, Ry, X)    LW Rtemp1, X
                   MOV Rtemp2, Rtemp1 / make a copy of X
                   BNE Rx, Rtemp1, exit / compare
                   MOV Rtemp1, Ry
                   MOV Ry, Rtemp2 / swap
Exit:              SW Rtemp1, X / store back X or Ry
```

CAS 是一个必须原子执行的复杂指令。上面给出的是基于微操作的一种可能实现, Rtemp1 和 Rtemp2 是内部寄存器 (非架构寄存器)。CAS 中的所有指令 (从 load 到 exit) 必须原子执行。因此, 在 CAS 退出之前, 所有指令必须执行完, 并且所有微操作必须打包成组整体退出。如果系统中有 store 缓冲, 那么 store 缓冲中的所有 store 必须在 cache 中先完成, 然后 CAS 中的 store 才能从 ROB 中退出并移到 store 缓冲中。

在 MSI-无效协议下, 确保 CAS 原子性的方法之一是等到 CAS 到达 ROB 顶部再执行。CAS 执行开始时, 会先获得包含 X 的数据块的一个独有的 (exclusive) 副本, 并且在 store 在 cache 中执行之前, 会一直保存在 cache 中 (在此期间不对总线请求进行应答)。这种实现机制的问题在于, LW 的值要等到 CAS 到达 ROB 顶部的时候才能返回。因此, 等待该结果 (Ry 的值) 的其他指令需要留在发射队列中, 一直等到 CAS 到达 ROB 顶部。

假如 LW 有 load 值撤回机制的保障, 并且在 store 成功完成之前 CAS 中的整个微操作序列会一直存在 ROB 中, 那么 CAS 也可以在到达 ROB 顶部之前进行推测执行 (除了其中的 store 操作)。任何时候, 如果 CAS 的 load 值必须撤回, 那么 CAS 的执行以及 ROB 中 CAS 之后的所有指令都必须进行回滚。对 CAS 的推测执行和基于 LL 和 SC 的实现非常类似。

利用 CAS 指令, 我们很容易在内存中实现一个循环队列 (类似 ROB 的硬件实现, 如图 7-36 所示)。出队和入队是两个基本的操作。“Top” 指针总是指向队列中最旧的一个插入项, 而 “Bottom” 总是指向下一个待插入项的位置。利用下面的代码, 入队和出队可以并行执行。如果没有 CAS 指令, 那么每次进行队列操作时, 都需要锁住整个队列, 这会降低并发性。下面的代码中, 队列大小是 256 个字节:

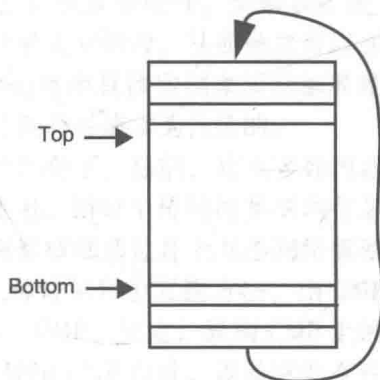


图 7-36

```
Enqueue (TOP,BOTTOM,R1){ /insert value at BOTTOM
  Begin: LW R2,TOP
        LW R3,BOTTOM
        SUBI R4,R3,#4          /Compute next value of BOTTOM
        ANDI R4,R4,0xFF        /MOD 256
        BEQ R2,R4,Begin        /TOP=BOTTOM-4 => Q is full
        CAS R3,R4,BOTTOM       /If BOTTOM is unchanged,
                               /decrement it atomically
        BNE R3,R4,Begin        /to grab queue entry
        SW R1,0(R3)            /Store entry at BOTTOM
  }
```

注意结尾的 store 是互斥完成的, 因为该位置已经被 CAS 指令原子保留了。

```
Dequeue (TOP,BOTTOM,R1){ /Remove value from TOP
  Begin: LW R2,TOP
        LW R3,BOTTOM
        BEQ R2,R3,Begin        /TOP=BOTTOM, Q is empty
        SUBI R4,R2,#4
        ANDI r4,r4,0xFF        /mod 256
        CAS R2,R4,TOP          /if TOP is unchanged
        BNE R2,R4,Begin        /decrement it atomically
        LW R1,0(R2)            /retrieve entry
  }
```

假设实现该队列的系统是弱顺序模型, 处理器支持推测乱序执行以及 CAS 指令的推测执行, 普通

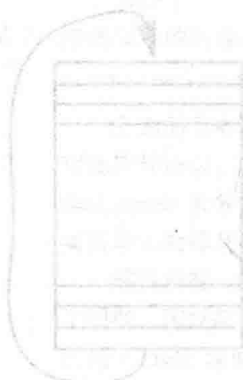
load 不受 load 值撤回机制保障, 但 CAS 中的 LW 在 CAS 执行之前仍受到 load 值撤回机制的保障。

(a) 假定有一个生产者和两个消费者, 并且场景如下: 队列为空, 并且两个消费者一直在试图执行出队操作。当生产者没有执行入队操作时, 请详细描述在两个消费者中将会发生什么, 入队和出队操作能否正确工作?

(b) 现在假定在 (a) 的场景中, 生产者往队列中插入了一个新值, 其中一个正在等待的消费者取到了这个值, 另一个消费者继续等待。请详细描述在三个处理器中将会发生什么, 入队和出队操作能否正确工作?

在回答上述问题时, 先假定 cache 协议采用的是 MSI-无效协议, 然后再假定是 MSI-写更新协议。请分别解释在这两种情况下, 需要如何来保证弱顺序模型下的推测执行是可行的。

(c) 当 CAS 不能推测执行, 而普通 load 可以推测执行, 并且不受 load 值撤回机制保障时, 会有什么变化? 这对性能有何影响?



片上多处理器

8.1 概述

本章我们主要讨论单芯片内的线程级并行。芯片内的并行存在几种不同的形式，在单核内可以同时执行多个线程以提高资源的利用率，这种方式称为核内多线程。根据从多个就绪线程中取指的方式和时机的不同，可以将核内多线程划分为三种类型：块式多线程，交错多线程，同时多线程。本章将给出支持这三种核内多线程所需的必要硬件补充和修改，从而使之可以在传统（单线程）的顺序或乱序上下文环境下正常工作运行。我们将用具体实例来展示细粒度多线程相比于粗粒度多线程的性能优势，当然这些优势是以额外的硬件成本为代价的。

紧接着介绍的是在片上集成多个处理器核来开发片上并行性的例子。目前，片上多处理器（CMP）系统从手机到数据中心服务器的各个计算领域广泛使用。相对于传统的共享内存多处理器（SMP），本书所讲解的 CMP 的主要优点源于其所有处理器核都通过片上互连网络紧密地集成在单个芯片上。我们将介绍当前用于构建 CMP 系统的 3 种常见片上互连方法，当 CMP 上的所有处理器核都相同时，我们称之为是同构（homogeneous）CMP，反之，异构 CMP 上的处理器核可以具备不同的功能，在本章中，将介绍不同类型的异构 CMP 设计，以及它们在性能和功能上的区别。

考虑到 CMP 在实际中广泛使用，对于未来的芯片设计者和程序开发者来说，有必要掌握不同的并行编程模型。要想发挥 CMP 的并行性，并行编程是一个需要解决好的关键问题。目前，OpenMP、Pthread 和其他一些并行编程模式已经帮助程序员快速地开发并行代码，但是，CMP 依然在硬件支持的并行和同步方面提供了一些特有的优化机会。例如，由于集成更加紧密，处理器核之间可以进行更快的数据通信，更高的带宽和较低的同步开销也使得我们可以使用带推测的新的同步和编程模型。在本章中，我们还将介绍一个基于事务内存（Transactional Memory, TM）的例子，这种并行编程模型减轻了如何选择最佳锁粒度的负担，TM 允许程序员通过将大量的指令序列组合成一个事务，从而使得线程在粗粒度级别进行交互。本章还对 TM 编程模型进行了描述，并给出了事务失败时，为协助事务提交和事务回滚所做的必要硬件修改。

并行化是一项非常有挑战性的任务。线程级推测执行（Thread-level speculation, TLS）是一种硬件辅助并行化的方法，该方法可以从串行程序中生成推测执行的并行线程，当推测失败时，TLS 需要额外的硬件支持回滚到之前的状态。本章中，我们将描述 TLS 是如何用于推测性的并行循环，在支持循环的推测执行时，需要额外的硬件来检测内存和寄存器的冲突。为了让读者对不断发展的自动并行化领域所面临的挑战和机遇有更深层次的理解，本章将详细描述相关硬件结构的演变。最后，我们简要介绍一些能够有效利用 CMP 上丰富线程资源的其他编程实例，如帮助线程（helper thread）可以加速非并行代码的执行，冗余线程（redundant thread）则可以提高系统的可靠性。

本章的主要内容如下：

- 8.2 节从技术和性能（并行性）的角度描述了片上多处理器研究的动机。
- 8.3 节主要讲述了不同粒度级别下的核内多线程，例如，块式多线程、交错多线程以及同时多线程。

- 8.4 节主要讲述了同构和异构处理器核组成的 CMP 架构, 包括对应的互连网络和 cache 架构。
- 8.5 节主要讲述编程模型——共享内存, 事务内存, 线程级推测执行, 帮助线程以及冗余线程等。

8.2 CMP 的基本原理

CMP 背后的驱动力主要源于对技术的需求, 自 2000 年以来, 单核处理器的性能提升已经越来越困难, 与此同时, 芯片上所能集成的晶体管数量还在持续增加, 在这种趋势下, 多处理器架构得以快速发展起来。倘若并行编程的问题能够很好解决, 那么 CMP 将有助于我们构建一个拥有海量线程并行性的大规模计算系统。

8.2.1 技术趋势

随着硅技术的发展, 单个芯片上集成数十亿晶体管已不再是问题, 片上多核处理器时代已经到来, CMP 将很快用于所有的计算领域: 从服务器到台式机, 再到移动设备。处理器核数则从双核开始, 按照摩尔定律发展, 未来有望呈指数增长。

在 CMP 时代之前, 晶体管数目的增长用于改善单线程的性能, 主要是用来开发指令级并行 (ILP), 具体的技术包括在流水线中支持更多的活动指令 (in-flight instruction)、更深的流水级和更加激进的推测执行等技术, 活动指令数的增长通常是通过增加微架构中相应部件的大小来实现的, 例如, Intel 在 1995 年推出的第一个乱序处理器 (Pentium-Pro) 中有 20 项保留站, 因此它最多允许 20 条 x86 指令并发执行, 而其 2000 年推出的 Pentium 4 处理器能够同时处理 128 条活动指令, 与之类似, 在一个周期内可以分发的指令数量也从 1 增长到 4, 为了在一个周期中可以执行多条指令, 寄存器堆、大的发射和 load/store 队列、大的重排序缓冲区等都需要配备多个读写端口, 随着这些结构的规模和端口数量的增长, 它们所占据的芯片面积更是呈平方关系增长。流水级方面, 越深的流水级就需要越好的分支预测器来减少错误分支预测所带来的开销, 而分支预测错误所引起的开销随着流水级深度的增加呈线性增长。此外, 为了解决内存墙 (memory-wall) 问题, 硬件支持的数据预取也成为经常采用的技术。所有上面提到的这些微架构的改进, 都会导致晶体管资源消耗的大幅增加。

历史上每一次硅工艺技术的进步都会伴随着电源电压的降低, 而电源电压的降低可以减少芯片功耗, 并且可以使晶体管内部的电场强度维持在安全范围内, 因此, 即使晶体管数量增加了, 但芯片功耗仍旧相对较低。但近年来, 电源电压降低的速度急剧放缓, 因此各代新工艺在降低功耗方面也越来越困难。

图 8-1 显示了功耗和性能 (使用标准 SPEC 测量得出) 之间的历史演化关系, 可以看到, Pentium 4 处理器的功耗超出 i486 处理器的 23 倍, 但其提供的性能仅比 i486 处理器高 6 倍。事实上, 我们发现功耗是按照 $f^{1.7}$ 的比例在增长, 其中 f 是芯片的工作频率。注意这幅图假定所有的处理器使用相同的制造工艺, 但是它考虑了随着时间的发展, 处理器架构和电路方面所提出的创新。很明显, 这种增长速率远远低于第 2 章式 (2.12) 中给出的理论功耗的增长比例 (f^3)。在第 2 章中介绍的计算性能的方法比较简单, 它只计算时钟频率的增长, 而没有考虑任何底层微架构或电路方面的创新。然而实际上, 每一代工艺的改进都会同时带来微架构方面的巨大改进, 进而提高处理器的性能。例如, Pentium 4 推出了乱序执行 (OoO), 使得处理器即使在 cache 失效后仍能继续执行, 此外, Pentium 4 还采用了扇区预取 (sector prefetching) 技术来将数据块预取到 cache, 从而提高内存系统的性能。与之类似, 在工艺改进时, 新处理器往往较上一代处理器都会在分支预测准确度方面有所提升。最后, 随着时间的推移, 还可以在结

构和电路级别上采用更好的节能技术。因此,考虑到所有的这些改进,其最终的效果就是,不同代微处理器的功耗以一个相对式(2.12)更低的速率在增长。

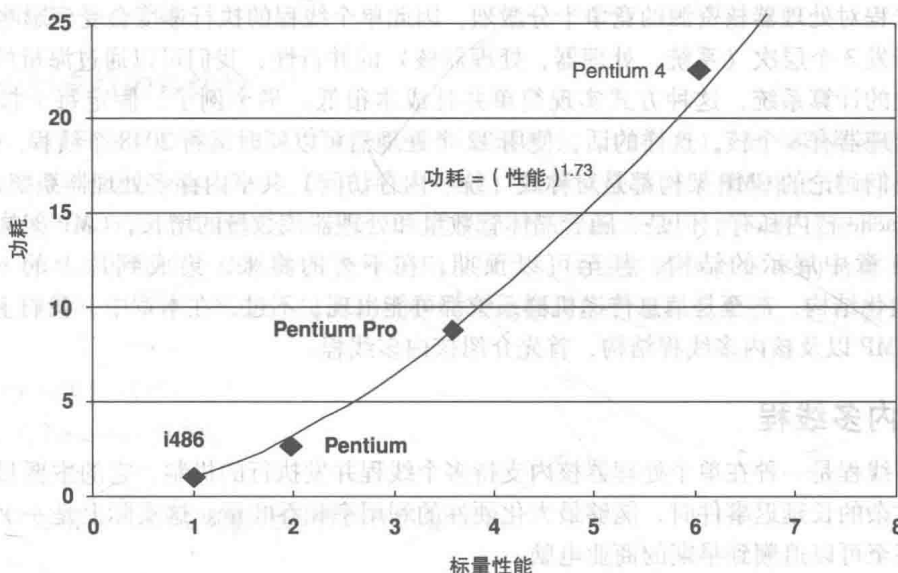


图 8-1 功耗和性能之间的历史演化关系

尽管 $f^{1.7}$ 的功耗增长速率相对于最坏情况下 (f^3) 的增长速率已经好很多了,但是这个增长速度仍旧很快。芯片行业已经意识到,在这种功耗增长速率下,继续开发 ILP 是不可持续的,此外,市场越来越倾向于提高硬件设备的可移动性,这会对处理器功耗的限制更加严格,这对技术也提出了更高的挑战。面对不断增长的晶体管数量,但却相对固定、甚至不断下降的功耗上限,芯片行业已经不得不减少 ILP 技术的开发,而转向线程级并行 (TLP) 技术,以此来提高微处理器的性能。

8.2.2 机遇

CMP 是中小规模的多处理器系统,但是它成本低,片上集成的功能完备,因此,它的出现使得以低成本和低复杂度构建一个海量线程的大规模多处理器系统的设想成为可能。多线程所带来的并行性可以在各个层面(系统、处理器和处理器核)上加以利用。在系统层面,TLP 的开发利用比 CMP 早很多年,它是在 SMP 系统环境下开发实现的,SMP 下的多个处理器芯片逻辑上共享物理内存,并且由专门的结构互连。SMP 系统在高性能计算(工程和科学应用)以及商业(数据库和 Web 应用)环境中可谓无处不在,而这些领域的应用本身也通常具有大量的、或易于开发的并行性。

SMP 和 CMP 的区别在于其集成水平,在 CMP 中,多个处理器核紧密集成在同一芯片上,这不仅实现了高效的内存共享,也实现了物理资源(如 cache 和协处理器)的共享,多个处理器核之间的紧密集成是通过片上互连网络实现的,这种片上互连网络实现时只是片上金属布线层的延伸,因此可以极大地降低处理器核之间的通信延迟,通常可以从 SMP 中的数百个时钟周期降为数十个时钟周期。CMP 中处理器核之间的通信一般是通过共享内存和共享 cache 来实现的。和 SMP 系统主要针对特定的高端应用领域不同,CMP 是面向大众市场的量产产品,它将并行计算的概念引入到了数字革命的方方面面中,包括个人电脑(PC)和移动设备等,在这类设备中以多进程、多任务等形式存在着大量并行性。

核内多线程进一步提高了并行性,它将处理器核资源在多个并发线程之间进行分时复用,

从而提高了处理器资源的利用效率。运行在同一个处理器核上的线程之间的通信大多通过共享内存以及处理器核内的一级 cache 来完成, 因此, 通信效率很高。核内多线程的主要缺点是, 由于多个线程对处理器核资源的竞争十分激烈, 因此单个线程的执行速度会受到影响。

通过开发 3 个层次 (系统, 处理器, 处理器核) 的并行性, 我们可以通过海量的线程并行来构建强大的计算系统, 这种方式实现简单并且成本很低。举个例子, 假定每个核有 8 个线程, 每个处理器有 8 个核, 这样的话, 使用 32 个处理器可以同时运行 2048 个线程。

目前我们讨论的 CMP 架构都是对称式 (统一内存访问) 共享内存多处理器系统, L2 cache 共享, L1 cache 核内私有。但是, 随着晶体管数量和处理器核数量的增长, CMP 架构将演变成类似于第 5 章中展示的结构, 甚至可以预期, 在不久的将来, 集成到片上的 cc- NUMA、COMA、层次化结构, 甚至是消息传递机群系统都可能出现。不过, 在本章中, 我们主要介绍现在主流的 CMP 以及核内多线程结构, 首先介绍核内多线程。

8.3 核内多线程

核内多线程是一种在单个处理器核内支持多个线程并发执行的机制, 它的主要目的是: 在面对各种复杂的长延迟事件时, 能够最大化硬件的利用率和吞吐量。这实际上是一个非常古老的思想, 甚至可以追溯到早期的商业电脑。

8.3.1 软件支持的多线程

计算资源一直以来都是非常珍贵的, 在 20 世纪 70 年代, 即早期计算时代, 计算资源的珍贵是由于计算系统的制造成本非常高。此外, 由于受到机械设备速度的限制, 诸如磁盘、磁鼓或磁带驱动器之类的 I/O 设备的访问速度一直比电子设备要慢好几个数量级, 因此, 从早期的单处理器系统开始, 分时操作系统就已经通过并行运行大量进程来提升计算资源的共享和分时复用, 其目标就是最大程度地把这些昂贵硬件资源利用起来。通常来讲, 硬件资源被用来做的有用工作越多, 那么单位时间和给定成本下所能完成的工作也就越多。

在分时系统中, 活跃的进程会被分配到内存资源中, 一旦准备就绪, 就可以运行。分配给活跃进程的典型内存资源是虚拟内存空间 (具体指的是一组页表) 和一个保存进程状态的进程控制块。就绪进程必须是活跃的, 且一旦处理器资源对其可用, 就可以立即执行, 当运行进程由于长时间的延迟操作而发生阻塞时, 例如碰到磁盘文件的读取, 页面错误, 或者是需要长时间等待的失效同步操作等, 操作系统会将阻塞进程切换出来, 并将它从就绪进程列表中删除, 重新放置到等待队列, 并从就绪列表中调度另一个进程执行。当正在运行的进程没有碰到任何长延迟事件时, 为了确保硬件资源的公平共享, 操作系统会以轮询的方式给每个运行进程分配一个时间片段或最大执行时间。

当碰到长延迟操作, 时间片耗尽, 或者接收到更高级别中断或异常时, 操作系统都会抢占当前运行进程, 这个过程通常称为上下文切换, 上下文切换需要以下几个步骤:

- 阻塞正在运行的进程, 并刷空流水线。
- 启动操作系统内核, 将当前进程的结构状态保存到进程控制块中, 并开始执行相应的中断处理程序。其中进程的结构状态包括寄存器、程序计数器以及其他状态寄存器 (如中断状态寄存器、条件码寄存器和页表基址寄存器等)。
- 内核恢复处理器中另一个已经就绪进程的架构状态, 并启动该进程执行。

通常情况下, 上下文切换需要数百个时钟周期, 因此, 操作系统只有在响应长时间的延迟事件时才会触发上下文切换, 以便分摊上下文信息保存和恢复的开销。

软件多线程仅需要少量的或完全不需要硬件支持, 实际上, 这种机制通常完全由软件来实

现, 比如在 Windows 和 Linux 内核中实现。软件上下文切换的好处在于高度灵活, 在上下文切换时, 操作系统可以确定哪些寄存器是进程真正修改了的, 然后选择性地保存和恢复那些在架构上可见状态的一个子集。

8.3.2 硬件支持的多线程

近年来, 随着处理器和内存速度之间的差距逐渐扩大 (内存墙), 处理器在等待内存事件完成时空闲的时间也越来越长。在 20 世纪 90 年代, 尽管乱序执行、推测执行和内存预取等技术在隐藏访存延迟方面确实起到了一定效果, 但由于当时 cache 失效的问题过于严重, 导致处理器资源 (比如功能部件) 依然难以充分利用。因此, 从提高指令吞吐率的角度来看, 当发生相对较短的延迟事件时, 暂时挂起当前执行线程也是有助于提高效率的。这些较短延迟事件包括:

- 失败的线程同步;
- L1 或 L2 cache 失效;
- TLB 失效;
- 长延迟指令 (如需要协处理器处理的指令);
- 异常。

需要注意的是, 从内核进行进程和线程调度的角度来讲, 暂时挂起的线程仍然是就绪的, 但是在处理器上真正调度它执行之前, 该线程还需要等待一个未完成的事件。硬件支持的多个线程动态共享一个处理器核的技术称为核内多线程, 处理器核中的每个线程都运行在一个硬件线程上下文环境中, 该环境包含一组能够支持单个运行线程执行和切换的硬件资源。操作系统内核将一个线程上下文看成是一个处理器资源, 也即将其看成一个单线程的处理器核, 因此支持核内多线程时, 操作系统并不需要做什么太大改动。在本章中我们使用下面这些术语:

- 当线程或进程已经分配到内存资源 (如页表) 时, 则称它是活跃的 (active), 活跃的线程可能处于阻塞 (blocked) 或就绪 (ready) 状态。
- 当活跃线程在等待操作系统事件 (如 I/O) 的完成或信号量的释放时, 则称它是阻塞的; 当一个线程已经准备好并且可以分配线程上下文时, 则称它是就绪的 (未阻塞的)。
- 当就绪线程分配到线程上下文, 且没有因为长时间延迟事件而被挂起时, 我们称之为执行线程 (running thread)。多个执行线程之间可以动态共享处理器核资源, 直到碰到长时间延迟事件, 此时, 由处理器核中的线程选择逻辑将其挂起。

例 8.1 五级流水线中 cache 失效延迟的影响 在单线程的五级流水线中, 当 cache 失效时, 处理器核的执行会被冻结住。举个例子, 假定每 20 个时钟周期会发生一次 cache 失效, 如果 L1 cache 失效, L2 cache 命中, 那么延迟为 20 个周期, 如果 L2 cache 也失效, 那么延迟为 200 个周期, 又假定每经过 200 个周期的计算之后会出现一次 L2 cache 失效。

图 8-2 给出 cache 失效的时间轴。假定无 cache 失效时的 CPI 是 1, 这意味着 L1 cache 中每条指令的失效数 (Misses Per Instructions, MPI) 是 0.05, L2 cache 的 MPI 是 0.005。如果考虑 cache 失效的延迟, 那么实际的 CPI 是多少?

处理器核以 20 个周期的时间间隔内轮流执行计算以及 L1 cache 失效, 反复执行 10 次, 直到碰到 L2 cache 失效, 然后处理器核暂停 200 个周期。因此, 执行 200 条指令的总时间为 $10 \times 40 + 200 = 600$ 个周期, 进而得到 CPI 为 $600/200 = 3$, 这个数字是不考虑 cache 失效时 CPI 的 3 倍。

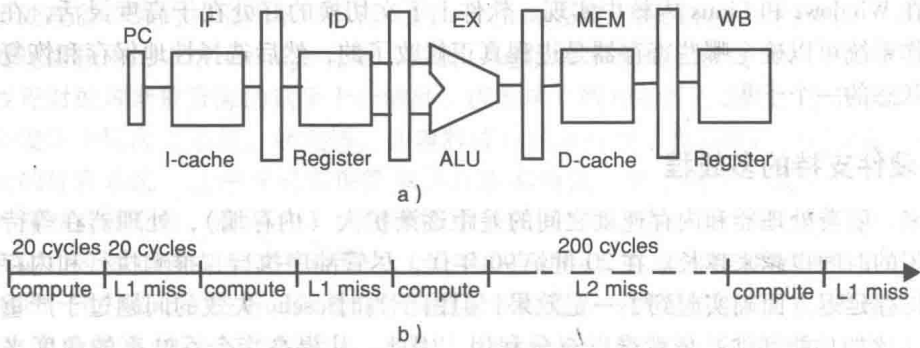


图 8-2 单线程五级流水线 (a) 和执行时间轴 (b)

例 8.1 中的数字并不夸张，可见，在等待内存的时候，处理器核浪费了三分之二的计算吞吐能力。在这段等待的时间里，处理器核本可以通过切换到其他的线程来执行的。但是传统的软件上下文并不适合这种场合，因为一级 cache 和二级 cache 失效引起的阻塞时间太短，都不足以抵消上下文切换引入的开销。认识到这一点后，现在的处理器在面对频繁发生的 cache 失效时，都在硬件上支持快速的上下文切换和高效的多线程。硬件支持机制应当避免每次出现 cache 失效时都进行机器状态的保存和恢复，并且，还应支持线程之间的快速切换机制。

硬件支持的核内多线程主要有两种基本类型：块式（粗粒度）多线程和交错（细粒度）多线程。

8.3.3 块式（粗粒度）多线程

在块式多线程中，只有当处理器核上的某个线程遇到长时间延迟事件时，才切换到其他线程执行，这种方式在某种程度上类似于软件多线程，但是规模不同。

五级流水线上的块式多线程

图 8-3a 给出了一种在五级流水线上支持两路块式多线程的方法，可以看到，增加的开销是比较适中的。在此架构中，当 L1 cache 失效发生时，处理器核并不阻塞，相反，在指令或数据 cache 失效时会引发一个低级别的硬件异常，该异常在写回阶段被处理，处理方式与常规五级流水线的软件异常处理机制相同，发生失效的指令以及处于 IF、ID、EX 和 MEM 阶段的所有指令都被刷掉，在这个周期结束时，发生失效的线程的程序计数器 (PC) 重置为那条 cache 失

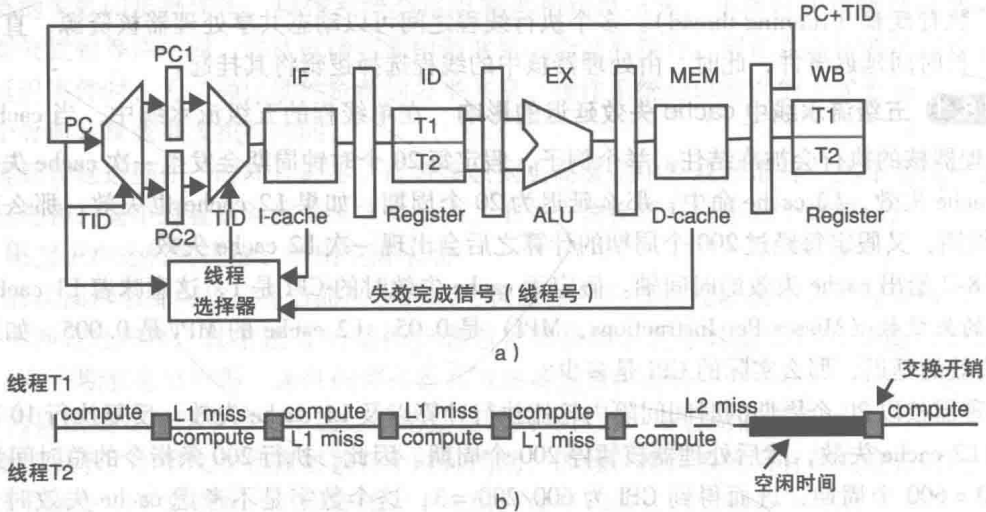


图 8-3 双线程的五级流水线 (a) 和执行时间轴 (b)

效指令的地址（请记住，每条指令需要带上自身的 PC，以便能从软件异常中恢复），与此同时，在 IF 阶段，线程选择器不再从出现失效的线程 PC 中进行选择，而是在下一个时钟周期中选择备用线程来进行取指，在此过程中，总共浪费了 5 个时钟周期。

除了支持清除流水线指令以及线程选择之外，还需要能够保存线程对应的架构状态，以便处理器核能从一个线程的状态迅速切换到另一个线程的状态。当进程或线程发生阻塞时，其架构状态保存在进程或线程控制块中，例如，一个两线程处理器核必须有两个程序计数器、两组寄存器、两个页基址寄存器以及两组条件/中断标志位。简单起见，图 8-3 只是显示了两份寄存器堆和程序计数器，每条指令中都带一个标识位，用于确定对应的线程，并能够选择其架构状态，这个标识位通常称为 TID，或线程上下文 ID。另一方面，L1 和 L2 cache 空间是在线程间共享的，为了避免线程间干扰导致的失效，两个线程的 TLB 表可以是彼此分离的。

在上面的简单实现机制中，控制取指的线程选择器也是非常简单的，它仅由底层的硬件异常机制驱动。线程只有在遇到触发硬件异常事件时才发生切换，当一个线程挂起，并且其他线程遇到长时间延迟事件时，挂起的线程会尝试重新开始执行，但是如果之前的失效处理尚未完成，那么重新执行时还会再次发生失效，此时线程会再次挂起。这样，两个线程会不断地重试执行，直到其中一个执行成功为止。如果我们使用更复杂的线程选择器的话（如图 8-3 所示），线程可能会一直停止住，直到 cache 失效已经处理完成，然后，线程被重新插入运行线程池中，并最终由线程选择器选定执行。在复杂的线程选择器中，也可以使用复杂一点的线程选择策略。

例 8.2 两路块式多线程五级流水中 cache 失效的影响 图 8-3b 给出了共享五级流水的两个线程的执行时间轴。线程 T1 执行 20 个周期（图中的 compute），直到它遇到 L1 cache 失效，处理需要 20 个周期的延迟，此时，线程 T1 挂起，线程 T2 开始执行，线程切换需要几个周期（称为“切换开销”）。线程 T1 和 T2 都不断在 compute 和 L1 cache 失效之间进行交替，这使得处理器核得以充分利用，因为它可以将一个线程的执行时间和另一个线程的 L1 cache 失效处理时间重叠在一起。此后，当线程 T1 遇到一个 200 周期延迟的 L2 cache 失效时，线程 T2 开始执行，但它也遇到 L1 cache 失效时，而且可能 L2 cache 中也同样失效，此时处理器核会出现空闲，因为线程 T1 和 T2 都在等待它们的 cache 失效处理完成，这段时间在图中记为“空闲时间”。当然，要支持这种方案的话，L1 和 L2 cache 必须是无锁（lockup-free）的，并且能够同时处理 2 个 cache 访问：可以是一个命中一个失效，也可以是两个都失效。

因为 L2 失效的延迟较长，为了提高处理器核利用率，需要提供更多的线程上下文来共享处理器核。随着线程数的增多，部分机制（如线程选择）也变得更加复杂，且线程占用的体系结构资源也随着线程数目的增长而增加，诸如 cache 和 TLB 这样的共享资源大小也必须等比例增加，以容纳线程数增多时的工作集。

图 8-3b 的执行时间轴只是为了便于解释而虚构的示意图。在实际的处理器核中，cache 失效或其他长延迟事件的发生时间是千变万化的，且这些事件的延迟时间也是可变的，这导致的结果是，可以重叠执行的时间段是不会像图中所展示的那样完美。例如，线程 T1 可能遇到 L1 cache 失效，然后线程 T2 被选中，并马上也发生失效，在这种情况下，两个 L1 cache 失效叠加在同一时刻了，此时处理器将处于闲置状态。处理器运行和发生 cache 失效次数的不同会导致最终性能的不同，但架构上的具体实现机制并没有区别。

乱序执行处理器核上的块式多线程

在五级流水线中，块式多线程很容易实现，存在的主要问题是切换的开销。这个开销限制了引起切换的事件必须是长延迟事件。由于处理器核通常无法在同一时刻执行多个线程的指

令，因此在进行线程切换时，必须清空掉流水线中属于被切换线程的所有指令。

清空流水线的成本主要包括两部分：(1) 按线程中的指令顺序提交在触发线程切换事件之前的所有指令；(2) 清空该事件之后的所有指令。一般来讲，在深度流水、乱序执行（Out-of-order, OoO）的处理器中，这个清空流水线的时间开销远远不止 5 个周期。在乱序执行的机器中，cache 失效异常会在重排序缓冲区的顶部被触发，后面的大量未提交指令需要清空，这导致大量工作被取消。然后下一个运行线程必须重填流水线。因此，切换开销很容易就达到几十个时钟周期。要在乱序执行处理器中使用块式多线程，除了那些保存架构状态的部件之外，还有一些其他部件也需要进行复制，例如，为了避免线程之间预测的干扰，分支预测器和 TLB 也可能需要复制多份。

例 8.3 支持两路块式多线程的乱序处理器上的执行情况 考虑两个线程片段，每个线程拥有 6 条指令的执行过程。图 8-4a 给出了两个线程片段的依赖图，图中展示了两个线程内指令的相关性和执行延迟。比如，线程 1 执行指令 X5 需要 1 个或 20 个周期，X5 在执行过程中需要进行一次访存，cache 命中则需要 1 个周期，cache 失效则需要 20 个周期。

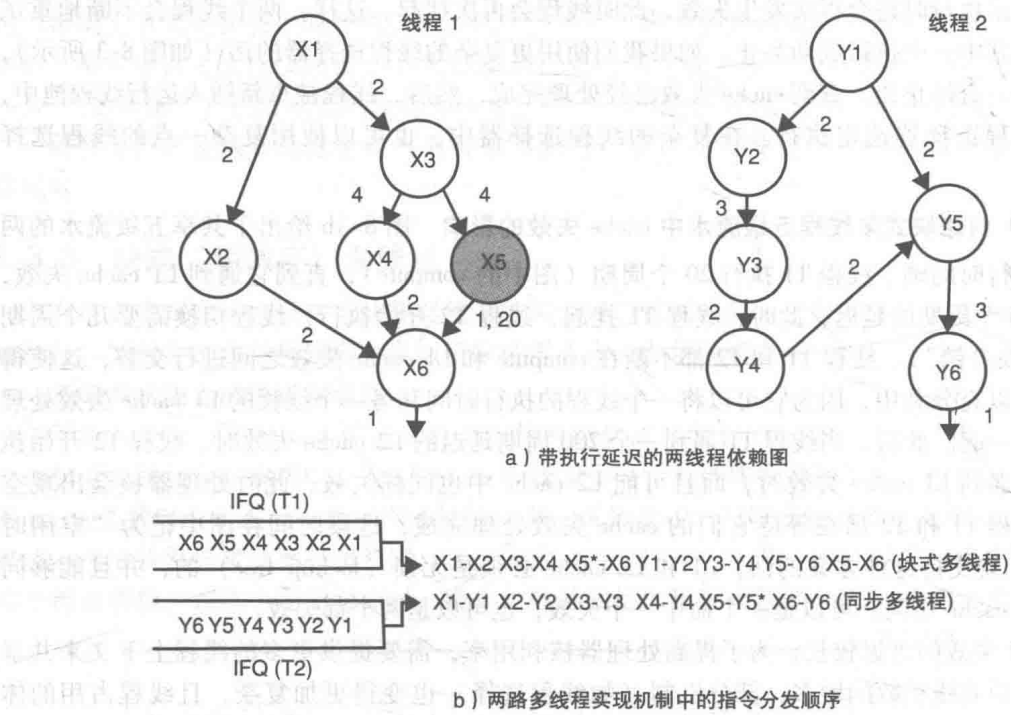


图 8-4 块式多线程和同时多线程中使用的线程示例

假定处理器是两路多线程、且带推测调度的乱序执行结构，每个线程都有自己的取指队列（IFQ），每个周期可以分别完成两条指令的取指、译码或者分发操作，每周期也可以最多提交两条指令，公共数据总线（CDB）每个周期可以前递两个结果数据。两个线程共享一个具有 4 项的发射队列，每个周期最多可以从中调度两条指令。寄存器端口和功能单元数量足够，不会引起结构冲突。基于上述假定，并且先执行线程 1，请给出这种情况下块式多线程的调度过程。

图 8-4b 显示了块式多线程的分发顺序。分发单元从两个 IFQ 中选出要分发的指令，在块式多线程的情况下，线程 1 的指令先被分发，然后再分发线程 2 的指令。在第一次执行时，X5 指令会引发一次线程切换。由于切换时线程 1 的 X5 和 X6 指令会被刷掉，因此当线程 2 执行完

成后，X5 和 X6 需要重新执行，当 X5 指令重新执行时，只需要一个时钟周期，具体调度情况见表 8-1。

表 8-1 块式多线程下的推测调度

指令 (延迟)	分发 (发射 Q)	发射	读寄存器	开始 执行	完成 执行	CDB	提交 (T1)	提交 (T2)
X1(2)	1(1)	2	3	4	5	6	7	
X2(2)	1(2)	4	5	6	7	8	9	
X3(4)	2(3)	4	5	6	9	10	11	
X4(2)	2(4)	8	9	10	11	12	13	
X5(1, 20)	9(3)	10	11	12*	12	13		
X6(1)	9(4)	11	12	13	13	14		
Y1(2)	15(1)	16	17	18	19	20		21
Y2(3)	15(2)	18	19	20	22	23		24
Y3(2)	16(3)	21	22	23	24	25		26
Y4(2)	16(4)	23	24	25	26	27		28
Y5(3)	24(3)	25	26	27	29	30		31
Y6(1)	24(4)	28	29	30	30	31		32
X5(1, 20)	32(3)	33	34	35	35	36	37	
X6(1)	32(3)	34	35	36	36	37	38	

在分发对应列中，括号内的数值表示在分发结束时，单个发射队列中被占用的项数，这个值不能大于 4（单个发射队列中的总项数），从指令分发到指令提交之前，相应项都需要保存在发射队列中。指令分发需要等待发射队列中有两个空项时再进行，以便可以将分发的两条指令对应项放入发射队列中。

开始时，处理器和单线程操作类似，直到线程 1 执行到指令 X5，此时，它检测到一个线程切换事件（在第 12 个周期），于是在第 13 个周期产生一个硬件异常，在第 14 个周期清空后端的流水级，在第 15 个周期开始从线程 2 分发指令，此时所有的发射队列项都是空的，在线程 2 执行完成后，线程 1 的指令 X5 和 X6 重新执行，在重新执行之前需要等待到第 32 个周期，此时线程 2 的所有指令都已经提交完成。

除了两次清空流水线和重新执行两条指令的开销之外，指令分发还常常受到其他因素的影响而阻塞，比如，由于线程内部的数据冲突，指令可能需要长时间占用发射队列项，这也会影响指令分发速度。在这个例子中，12 条指令的发射需要 32（即 34 - 2）个周期，发射速度为每个周期 $12/32 = 0.375$ 条指令，即 CPI 约为 2.7，这个 CPI 值远远好于单线程处理器核对应的值。在单线程处理器核中，第一个线程需要处理完 cache 失效后，第二个线程才能执行。

支持块式多线程的处理器实例

块式多线程在短流水线和顺序执行的机器中都已经实现。IBM 的 iSeries SSTAR 处理器就是块式多线程处理器的例子，IBM 将此特征称为硬件多线程（HMT）。SStar 是用于运行商业应用负载的服务器处理器，它是一个五级流水线四路超标量的顺序执行处理器。由于商业应用负载线程数多，但其 cache 失效率非常高，因此 HMT 主要在 L1 或 L2 cache 失效时切换线程。为了最小化处理器规模和设计复杂度，HTM 只支持两个线程：一个前台线程和一个后台线程。控制寄存器指定了线程上下文切换的条件，比如 L1 cache 失效、L2 cache 失效以及线程切换超时等。在后一种情况下，即使线程没有遇到 cache 失效，它也会在预定时间后强制放弃对处理器

核的占用，这种策略为线程之间的公平性提供了保障。

当前台线程遇到 L1 cache 失效时发生线程切换，此时后台线程有可能已经在等待另一个 cache 失效；如果后台线程正在等待 L2 cache 失效，那么前台线程此时将不会挂起，而是先阻塞流水线，直到数据从 L1 cache 返回。如果前台线程也遇到 L2 cache 失效，则会进行线程切换，即使这种情景下两个线程都需要等待各自 L2 cache 失效的完成。每个线程保存其自身的架构状态，但功能单元、流水线资源、cache 和 TLB 等资源是所有线程共享的，当发生 L1 Dcache 失效时，线程在写回阶段发生切换。由于后台线程的指令在其对应的指令缓冲区中已经缓存好并完成预译码，线程切换的开销只有三个周期。HMT 通过最小化线程独立的状态副本和最大化资源共享，使得处理器规模开销控制在 5% 以下，且对时钟频率的影响小于 1%。

块式多线程的另一个例子是 Intel 的 Montecito 处理器，这是一个双核结构，每个处理器核有两个线程，支持 IA-64 (Itanium 2) 指令集架构 (EPIC 架构)。每个处理器核都是顺序执行，长延迟的 L3 cache 失效将导致片外的存储访问，这种长延迟很难通过线程内的其他就绪指令来隐藏，因此，Montecito 在每个处理器核中使用块式多线程，当发生诸如 L3 cache 失效/数据回填、时间片耗尽、软件给出线程切换提示等事件时，处理器核进行线程切换。线程切换提示是由添加到 ISA 中的特殊指令来给出，并引导线程在执行过程中放弃处理器核占用。这类事件会通知到线程选择器，并分配不同的线程紧迫级别，具有最高紧迫级别的线程总是能够被选中执行。因此，当暂停线程的紧迫级别比正在运行线程的紧迫级别更高的时候，就发生线程切换。每个线程都有一份寄存器堆的副本，但是共享各层次存储结构以及分支预测部件。为了防止线程的分支历史被其他线程污染，进行分支历史和分支预测表索引时，需要包含线程 ID。实现多线程所需增加的机器架构状态副本导致芯片面积增加了 2% 左右。

8.3.4 交错（细粒度）多线程

粗粒度多线程结构下同一时刻只有一个线程在运行，因此很难充分利用由于指令操作所引发的短延迟阻塞。

在交错多线程中，多个线程同时在处理器核上运行。处理器在连续几个周期内可以对来自不同线程的指令进行取指、译码和调度。因此，来自不同线程的指令可以以细粒度交错的方式执行。如果线程遇到长时间的延迟事件（如 cache 失效），则会暂时挂起，通过每个时钟周期交错执行不同线程，即使某个线程中存在数据依赖和冲突，处理器也有可能一直处于工作状态。

五级流水线上的交错多线程

图 8-5 给出了在支持交错多线程的五级流水线上运行 2 个和 5 个线程的具体操作情况。在每个周期中，依次选择运行线程执行，以便不同线程占据不同的流水阶段。当运行两个线程时，由于同一线程的指令每隔一个周期启动执行一次，因此，即使 load 指令后紧跟一条有依赖的指令，也不会引起流水线阻塞。当运行 5 个线程时，每个流水级处理不同线程的指令，各个流水级之间可以互不影响。

支持交错多线程的五级流水线基本架构与图 8-3a 中的结构也有点相似，但是由于不同线程的指令在流水线上并发执行，与之相比有以下三个显著的差别：

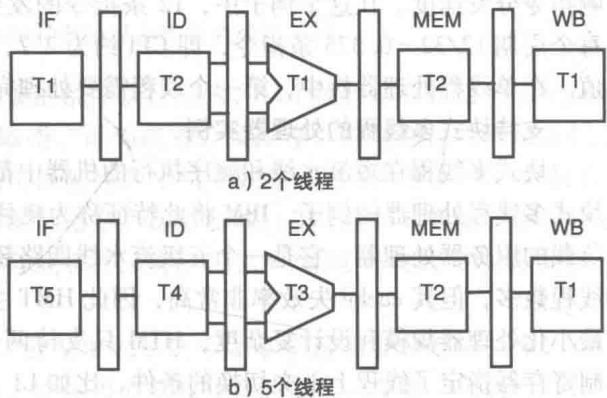


图 8-5 五级流水线中的交错多线程

- 数据前递必须是线程可感知的,这意味着任何前递值都必须携带目标线程的 TID,只有当前递值携带的 TID 与开始执行指令的 TID 相匹配时,才能将该值正确前递给同一线程。
- 流水级清空也必须是线程可感知的。当写回阶段接收到异常时,IF、ID、EX 和 MEM 流水级中的数据不能直接清空,包含其他线程指令的流水级不可以清空,同样道理,当出现分支跳转时也是类似的情况。
- 线程选择算法有所不同。线程选择器每周期都需要选择一个不同的线程,可用的算法有很多种,一种简单的算法是以轮询的方式从运行线程集中选择下一个线程,每当遇到长时间延迟事件时,线程选择器就将当前线程挂起,并暂时将它从运行线程集合中删除,一旦长延迟事件完成,线程选择器将其重新插入到运行线程集合中。

五级流水线中的交错多线程比块式多线程效率更高,这是因为前者可以消除流水线中的气泡,而且发生长延迟事件和分支跳转时,只需清空少量指令,图 8-5 分别以 2 个线程(T1 和 T2)和 5 个线程(T1 ~ T5)的运行情况进行了说明。当运行 2 个线程时,分支跳转的代价是 1 个时钟周期,硬件异常或软件异常的代价是 3 个时钟周期。而当运行 5 个线程时,分支跳转没有额外代价,异常的代价仅为 1 个时钟周期。

需要注意的是,当单线程程序运行在多线程处理器(块式多线程或者交错多线程时)上时,其速度和运行在基本的单线程五级流水线上是一样的。

交错多线程的实例

交错多线程微架构的一个很好的例子是 Sun Sparc T1 和 T2 处理器核架构,T1 和 T2 的主要目标市场是商业或客户端-服务器应用,如数据库系统、联机事务处理以及 Web 应用等。这些应用都是大量的客户端向一个集中的服务提供者请求服务,在这种情况下,客户端请求会在服务器上触发多个独立的并发执行的线程进行处理,这些线程只执行少量的或完全没有浮点指令,它们主要执行整数和内存访问指令,且 cache 失效开销占据了执行时间的大部分。客户端-服务器应用由于具有大量固有的并行性,需要很好的线程执行吞吐量,因此,这类应用是多线程处理器的主要目标。当几个线程同时在处理器核上运行时,通过在 cache 失效时执行不同线程的代码,可以有效保证处理器核利用率和吞吐量维持在较高水平。

Sun Sparc T1 有 8 个处理器核,每个处理器核可以运行 4 个线程,共计 32 个线程,Sun Sparc T2 有 8 个处理器核,每个处理器核可以运行 8 个线程,因此总共有 64 个线程。处理器核配备的是单发射的整型流水线,T1 和 T2 中,每个处理器核都有自己的 L1 cache,处理器核通过交叉开关共享一个分体的 L2 cache,并通过 L2 cache 中的目录结构来维护 L1 cache 之间的一致性。

图 8-6 给出了 T1 的基本处理器核架构,方便起见,假定每个处理器核有两个线程。由于每个线程的指令还是按照线程序开始执行,因此,这还是一个顺序流水线结构,类似之前介绍的五级流水线。不过,这个结构更加复杂,当碰到长延迟事件时,不需要采用硬件异常将线程挂起。它与简单的五级流水线的主要区别是,降低了译码和取指这两个流水级的耦合性,这两个流水级通过两个独立的取指队列(IFQ)进行隔离,每个线程一个取指队列,其目的是当发生指令 cache 失效时,也能够较好地均衡指令发射到流水线中的速度。而分支预测使用的是 Sparc V9 分支指令中提供的软件预测(暗示)位来进行静态预测。

每个周期,线程选择逻辑选择一个线程进行取指和译码,如果两个线程都在运行,且都执行的是低延迟的指令,那么线程选择算法只是简单地轮询,如果其中一个线程遇到长延迟事件,那么线程选择逻辑将暂时挂起该线程(不再选择该线程执行)。除此之外,线程选择算法也能通过操作码获取一些指令的延迟,并据此调整它们的选择。当发生分支跳转、浮点运算或复杂整数运算(如除法或乘法)时,可能会暂时取消选择该线程调度执行。

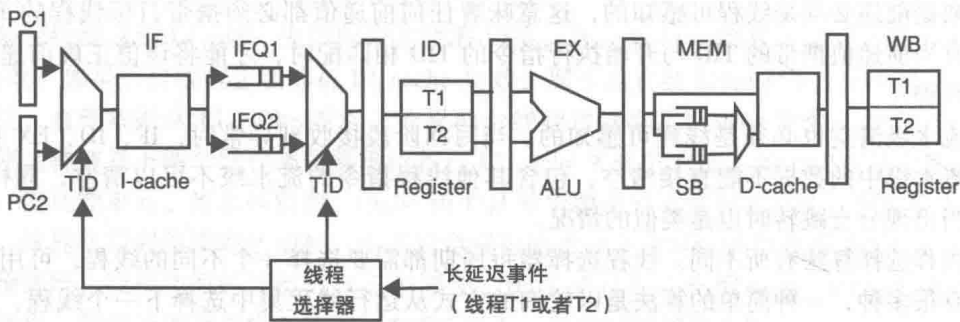


图 8-6 支持两路交错多线程的顺序流水线

桶形处理器

极端情况下，交错多线程变成一种非常简单的流水线结构，称为桶形处理器（barrel processor）。由于有足够的线程上下文和运行线程，即使在同一时刻，每个线程都仅有一条指令存在于流水线中，处理器核也依旧能够保持忙碌状态，因此也就无须用于消除数据和控制冲突的硬件结构了。类似于单周期的非流水处理器结构，这里的每个线程上下文一次也只执行一条指令，但是只要有足够的运行线程，那么它就可以忍受长时间的内存访问延迟，可以使用容量大速度慢的数据 cache 结构，甚至完全不用 cache，都不会导致流水线出现饥饿（starving）。图 8-7 给出了一个五级流水线的桶形处理器结构，该结构通过复制 32 个线程上下文实现了对 32 个线程并行执行的支持。

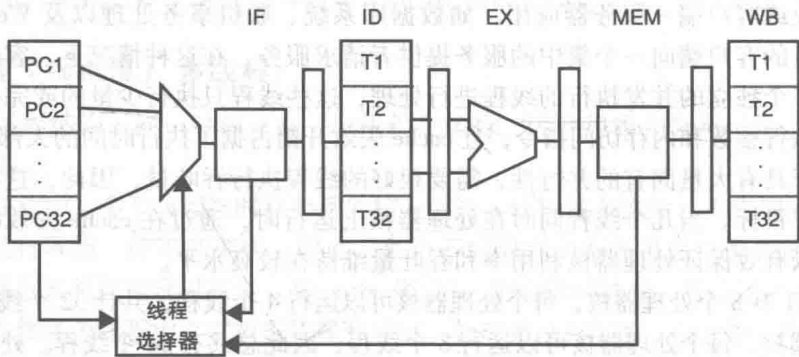


图 8-7 桶形处理器

这种架构尽管概念上很简单，但至今也还没有成功的范例，主要原因如下：

首先，在目前的技术条件下，如果要保证流水线不空闲，则需要数以百计的线程，而大量线程上下文所需的硬件开销是非常可观的。

其次，这种微架构与过去传统的处理器微架构差别太大，对存在大量并发线程的工作负载来说，利用廉价的现成微处理器搭建的大规模并行架构在市场上更加成功。

最后，即使这种机器具有非常高的吞吐率，单位时间内可以执行数百个线程，但是单个线程的执行速度却非常慢。除了那些具有无限并行性的应用之外，多线程处理器通常不能为了线程吞吐量的提升而完全牺牲单线程的执行时间。

桶形处理器的实例

桶形处理器背后的技术思想很早就出现了，事实上，早在 20 世纪 60 年代，控制数据公司（CDC）率先在 CDC6600 的 I/O 处理器设计中实现了这一想法，之后，在 80 年代初，Denelcor 公司推出了包含多达 16 个处理器单元的多处理器系统 HEP（Heterogeneous Element Processor），

其中每个处理器单元都是八级流水。在任何时刻,不同的流水级处理的是来自不同进程的指令,因此,为了充分利用流水线,至少需要8个进程。由于不同流水级中的指令都是相互独立的,并且每个进程一次只执行一条指令,不存在数据和控制冲突,因此无需进行数据前递、冲突检测、流水线阻塞以及清空等操作。处理器也没有 cache,直接通过交换网络访问内存系统。当然,那时处理器和内存之间的速度差距并没有今天这么大,每个处理器的最大指令吞吐量是1.25MIPS,因此8个处理器的最大指令吞吐量总和是10MIPS。

后来,Denelcor公司的HEP发展成TERA,这是一个由256个处理器节点组成的多处理器系统,TERA中的处理器架构被称为“地平线(Horizon)”结构。Horizon指令由三个RISC操作组成,每个操作类似于MIPS指令,因此Horizon结构也可以看成是一种超标量处理器或长指令字(LIW)处理器,每个处理器节点都是流水结构,包含指令存储和cache,支持128个指令流(一个指令流等同于TERA中的一个线程),硬件支持程序计数器和寄存器堆的复制,128个指令流总共有128个程序计数器和4096个寄存器。每个周期从就绪指令流池中选择不同指令流中的指令执行,如果指令流的指令与之前发射的、现在还在流水线中的指令都没有数据依赖关系,则该指令可以发射。TERA没有针对数据冲突的硬件支持,为了跟踪依赖关系,Horizon的每条指令中增加了一个前看(lookahead)字段,表示后续有多少条指令可以无需等待当前指令完成就可以发射。在相同指令流中,由前看字段标明不存在数据依赖的那些指令,就可以在流水线上并发执行。发射之前还需要满足如下要求:(1)指令是有效的;(2)指令的寄存器访问没有结构冲突。数据内存分布于多处理器的不同节点上,并且处理器上没有数据cache,因此也无需考虑cache一致性。内存的访问时间可变,从50~80个时钟周期不等,具体取决于数据所在内存单元的位置。由于支持128个硬件指令流和大量的软件线程,因此,即使某些指令流的指令执行需要80个周期,流水线也可能保持100%的利用率。

8.3.5 乱序执行处理器上的同时多线程

块式多线程在推测乱序处理器核上的效果并不好,因为线程切换的开销很大。在块式多线程中,单个周期内处理器核不能同时支持多个线程并行执行,新的线程必须在旧线程完全清空流水线后,才能开始执行。

交错多线程适用于乱序处理器核,而同时多线程(SMT)是它的进一步延伸。随着复制的资源越来越多,并假定处理器核每周期都可以并行执行多个线程,那么线程就可以进行细粒度的交错执行,就像在简单处理器核上的交错多线程实现一样。和块式多线程不同,同时多线程下的长延迟事件不被当作硬件异常来处理,而是会将指令分发单元重定向到从其他线程开始分发指令,在这种情况下,流水线后端不会因为线程切换而清空,引起线程挂起的指令仍旧保留在流水线中,直到长延迟事件完成,该指令执行结束退出流水线。

在SMT的每个周期,都会从不同的线程调度处理器核指令。在超标量处理器核(每个周期可以分发多条指令)上下文中,来自不同线程的指令可能在同一周期进行取指、译码和调度,从而使得处理器核共享粒度比交错多线程更细。在每个周期,来自不同线程的指令从一开始就可以争夺共享资源,从而有助于隐藏由于短操作延迟和资源共享所引发的阻塞,进而提高硬件利用率和指令吞吐量。

除了架构状态,SMT通常还需要为每个线程复制取指队列、重排序缓冲区以及load/store队列。此外,指令调度、数据前递和指令清空必须是线程可感知的(thread-aware),每条指令在执行过程中,必须携带自身的线程上下文ID。

- 数据前递必须是线程可感知的。

- 流水级清空必须是线程可感知的。当由于错误的分支预测或异常而清空流水线后端时，包含来自其他线程指令的流水级并不清空。在流水级中，可以通过将指令的 TID 与引起清空操作的指令 TID 进行匹配来判断；同样，前端流水级中的结构也只有与线程匹配时才被清空。
- 指令调度必须是线程可感知的。发射队列中的项必须有 TID 标记，并且，当结果需要读取发射队列中某条指令的操作数时，结果对应的 TID 和等待发射指令时 TID 必须匹配。

在超标量处理器中，如果同一周期可以从不同线程取指、译码和调度，那么可以大大简化硬件复杂度。我们已经看到，在超标量机器上如果只运行单个线程，那么取指和分发阶段（重命名操作执行的阶段）将是严重瓶颈。在取指阶段，由于同一线程的一串连续指令中可能存在条件跳转，这会导致在一个周期内无法同时取太多指令。同样由于单一指令流中存在寄存器依赖，如果对大量指令进行重命名，则逻辑操作非常复杂，需要对同一个重命名表进行大量的访问。在变长指令宽度的架构中，例如 Intel 的 x86，即使对同一线程的连续多条指令译码也十分复杂，译码器可能需要先对第一条指令进行局部译码，然后才能识别出第二条指令的起始地址。在取指、译码和重命名阶段的依赖检查会导致串行瓶颈，或者增加处理器的时钟周期时间。

在同一周期内，通过交错执行来自不同线程的指令，SMT 结构可以有效消除上述串行瓶颈。这是因为在同一周期，可以使用多个程序计数器对来自不同指令流的指令进行取指、译码和调度。例如，运行单线程的 8 路超标量机处理器必须从同一线程取 8 条连续的指令，然而，如果有 SMT 支持，那么每个周期只需要从 4 个不同的线程分别取指、译码和重命名 2 条指令，这无疑更容易实现。

例 8.4 支持同时多线程的两路乱序处理器的执行情况 我们还是使用例 8.3 的场景，但是考虑支持同时多线程，并且同一周期可以从不同线程灵活分发指令，这样，通过细粒度共享处理器核资源的方式提高了硬件的利用率。具体调度过程如表 8-2 所示，每个周期可以从每个线程分发一条指令，在第 18 个周期，与 X5 指令相关的长延迟事件导致线程 1 被挂起，这时 X6 指令已经分发出去，因此 X5 和 X6 两条指令都各自占据发射队列的一项。这样，在 17 个周期（19-2）内，总共发射了 11 条指令，因此，这个例子中的发射率是每个周期 $11/17 = 0.65$ 条指令，即 CPI 为 1.55。

表 8-2 同时多线程下的推测调度

指令 (延迟)	分发	发射	读寄存器	开始 执行	完成 执行	CDB	提交 (T1)	提交 (T2)
X1(2)	1(1)	2	3	4	5	6	7	
Y1(2)	1(2)	2	3	4	5	6		7
X2(2)	2(3)	4	5	6	7	8	9	
Y2(3)	2(4)	4	5	6	8	9		10
X3(4)	7(3)	8	9	10	13	14	15	
Y3(2)	7(4)	8	9	10	11	12		13
X4(2)	10(3)	12	13	14	15	16	17	
Y4(2)	10(4)	11	12	13	14	15		16
X5(1, 20)	15(2)	16	17	18*	37	38	39	
Y5(3)	15(3)	16	17	18	20	21		22
X6(1)	17(2)	36	37	38	38	39	40	
Y6(1)	17(3)	19	20	21	21	22		23

同时多线程的实例

近年来,包括 IBM 和 Intel 在内的多家公司都已经在他们的处理器核微架构中实现了 SMT,但是所有实现都限定每个处理器核只支持两个线程,并且也还没有实现真正意义上的 SMT。真正意义上的 SMT 要求,可以在同一周期内对来自不同线程的指令进行取指、译码和调度。在乱序执行处理器中,同时多线程通常基于交错多线程实现,因为这两类多线程都需要相同的线程感知硬件支持来实现指令的转发、清空和调度。在商业系统中,使用这种保守方法的主要原因在于:我们希望即使只运行单线程,也能够利用所有的处理器核资源。

Intel 在 NetBurst (Pentium 4) 的微架构上通过超线程技术 (HTT) 实现了 SMT 机制,后来,也在 Atom 和酷睿的一些微架构中进行了实现。在 HTT 中,逻辑处理器 (又称为线程上下文) 几乎共享物理处理器的所有资源,包括 cache、物理寄存器、功能单元、分支预测器、控制逻辑和总线等。相反,寄存器别名表 (RAT)、下一条指令指针 (类似于程序计数器)、返回堆栈预测器 (预测间接跳转的目标地址)、取指队列、微指令队列、踪迹缓存填充缓冲区、指令 TLB、重排序缓冲区和 store 缓冲区等都需要进行复制备份。诸如发射队列这样的一些资源还可以通过阈值策略来灵活决定是共享还是进行资源复制。比如,不同线程的微指令可以先通过轮询的方式在发射队列中分配对应项,当某个线程对发射队列的占用达到阈值时,就不再允许该线程的微指令继续分发到这个发射队列中。

IBM 的 Power 5 处理器在 Power 4 微架构基础上实现了 SMT。Power 4 的每个处理器有两个五路乱序执行的处理器核。分支预测后,每个周期可以从指令 cache 中取出 8 条指令,并将这些指令发送到 IFQ 队尾。在 IFQ 队首,每个周期最多可以取出 5 条指令组成一组,构成一个分发单元。不同的组之间通过组完成表 (GCT) 来维持相互顺序,其作用类似于 Power4 中的 ROB。每个组由连续指令动态生成,一旦组内的所有指令都不存在结构冲突了,就可以按照进程一次分发一组指令。需特别处理的是,load 和 store 指令必须在 load 请求队列 (LRQ) 或 store 请求队列 (SRQ) 中保留一项,LRQ 和 SRQ 的总长度都是 32 项,它们构成了对应的 load/store 队列,可以用于检测内存访问冲突。组内的指令乱序发射到 8 个执行单元,每个周期最多可以发射 8 条指令。GCT 会跟踪记录组内指令的完成情况,并按照进程依次提交不同的指令组,因此,异常只能在不同的指令组边界被接收。此外,逻辑寄存器 (架构寄存器) 需要映射到真正的物理寄存器,因为两者数量可能不相等,比如,某种情况下,逻辑通用寄存器 (GPR) 有 36 个,而物理通用寄存器则可能有 80 个。

除了支持两个硬件线程上下文之外,Power 5 的处理器核架构与 Power 4 的基本相同。Power 5 处理器核既支持 SMT 模式,也支持单线程模式,这两种情况都是从同一个线程进行取指、译码和调度。每个线程都有一个独立的 IFQ,每周调度硬件都会基于优先级选择算法从两个 IFQ 队列中选择一个进行译码,然后组成一个最多 5 条指令的指令组,一旦这个指令组所需的资源准备好了,就可以马上分发。接下来是对寄存器进行重命名的操作,Power 5 处理器核有 120 个物理通用寄存器和 120 个物理浮点寄存器,在 SMT 模式下,两个线程动态共享这两个寄存器堆,不同线程的寄存器通过在结构寄存器号上添加一个线程位来进行区分。在单线程模式下,所有的物理寄存器都分配给单一线程;而在多线程模式下,当分发一组指令时,将从共享 GCT 的 20 项中分配一个给对应的指令组,来自两个线程的指令组按照它们各自的程序序交错保存在 GCT 中。指令组中的指令可以乱序发射和执行,发射队列由两个线程共享。一旦一组指令执行完成且没有出现异常,并且该组指令是线程程序中最早的一组,那么这组指令就可以退出。每个线程都有一个组完成单元,因此每个周期都可以有两组指令退出。为了支持两路 SMT,Power 5 将 Power 4 中的 LRQ 和 SRQ 拆分成两半,每部分各 16 项。

在 SMT 模式下, Power 5 处理器核对指令的译码和分发进行了节流控制, 以避免某一个线程独占像发射队列这样的共享资源, 具体的节流控制措施包括以下 3 个机制:

- 给每个线程设置不同的优先级。可以通过软件或硬件动态地设置优先级, 高优先级线程的指令可以被优先选择并译码。例如, 当某个线程占据的 GCT 项超过预设阈值时, 可以使用这种机制。
- 停止选择线程译码。当某个线程的未完成 L2 失效数超过预设阈值时, 它会对发射队列造成阻塞, 并使得其他线程的执行速度减慢, 甚至完全停止。因此, 当检测到这种状况时, 会暂停该线程的指令译码和分发, 直到该状况解除。
- 清空 IFQ 中某个线程的指令。当某个线程执行到非常长延迟的指令时 (如发生同步 acquire 操作失败), 就会采取这种极端的措施。

线程优先级也可以被用来减缓那些无效操作的线程, 或者加快那些紧迫性比较高的线程。例如, 当一个线程处于空循环操作或在忙等锁时, 应当降低其优先级, 或者, 如果线程是带有 deadline 的实时应用, 那么应当提高其优先级。在 Power 5 中, 每个线程有 8 个优先级 (从 0~7), 不同的优先级可以对线程的译码速度进行控制。例如, 当优先级相等时, 两个线程的译码速度相同, 当两个线程的优先级相差 7 时, 则只有一个线程在译码, 从而导致其效果就是单线程执行。

近期的 Power 7 处理器是一个配置 8 个乱序处理器核的 CMP 架构, 每个核每周期可以取指并发出 8 条指令, 并最多分发 6 条指令。在 SMT 模式下, 每个处理器核可以并发执行 4 个线程, 因此每个处理器的线程上下文总数为 32。

最后, 在 Sun 公司的 16 核 Niagara 3 处理器上, 每个 SPARC 处理器核可以支持 8 个线程的执行, 因此片上总共可以支持 128 个线程。

8.4 片上多处理器架构

传统的 CMP 分为多核 (片上不超过 8 个核) 和众核 (超过 8 个核), 可以预计在不久的将来, 所有的 CMP 都将是众核结构的。CMP 可能是同构的, 也可能是异构的, 这取决于处理器核是否相同。处理器核可能是多线程的, 也可能是单线程的。此外, 由于处理器核之间距离很近, 因此也可以高效地共享一些硬件资源。

CMP 系统具有如下几个理想特性:

- 设计简单。设计者只需要构建一个处理器核, 然后将其复制多份就可以充分利用不断增加的片上晶体管资源, 从而极大地降低了芯片设计的复杂性。此外, 芯片的测试和验证也同样得到了简化。
- 功耗更可扩展。随着处理器核数的增加, 片上总功耗呈线性增加, 这使得 CMP 结构比复杂的单处理器结构更具吸引力。比如, 将处理器核数增加一倍, 功耗也只增加一倍, 而如果我们将其频率提高一倍, 那么功耗将变为原来的 8 倍。
- 通信延迟低。当多个处理器核集成在单一芯片上时, 核间的通信延迟非常低。低延迟的核间通信也导致了新的并行编程模式的出现, 其中一个例子是细粒度线程并行。在 CMP 时代之前, 编译器和应用程序开发人员都不得不开发粗粒度并行线程以利用 SMP 系统中的并行性。在两次通信之间, 每个线程需要执行上百条, 甚至上千条指令。线程之间也不能频繁地共享数据, 因为共享数据需要通过片外通信来实现, 开销很大。而 CMP 允许编译器和应用程序开发者更加高效地利用并行性, 可以开发频繁访问共享数据以及长度更短的线程。
- 模块化和可定制。基于 CMP 架构, 可以通过改变片上处理器核数来针对不同市场需求

定制芯片。例如，对嵌入式系统市场来说，双核芯片就可以满足需求，而面向高性能应用市场，则需要众核处理器芯片。这样，CMP 系统通过简单的核数变换，就能实现针对特定市场的定制化。

8.4.1 同构 CMP 架构

在同构 CMP 架构中，所有处理器核都是相同的。图 8-8 展示了一个简单的四核 CMP 架构，4 个处理器核完全相同并且结构对称，因此，设计者只需要设计一个处理器核，然后再对其进行复制，就能形成 CMP 结构。这些处理器核可能是简单的五级流水或者复杂的乱序处理器核，每个核都有一个私有的 L1 cache，图中，核间共享一组 L2 cache（4 个 bank）。尽管有些 CMP 中，L2 cache 是私有的，L3 cache（对应图中结构的 L3 是在片外）是共享的，但是共享 L2 cache 仍是 CMP 最主流的结构。

基于总线的 CMP

图 8-8 中的片上 4 个处理器核是通过总线连接的，并且共享 L2 cache 组织成 4 个 bank（存储体），这个结构类似于 SMP 或 UMA 共享内存机器，只是共享的 L2 cache 取代了共享内存模块。例如，地址为 N 的存储块被固定存放在第 k 个 L2 bank 上，其中 $k = N \bmod 4$ 。L1 cache 之间的一致性通过侦听协议来维护，侦听协议借助于总线的广播和总呼（broadcast）功能实现。共享 L2 cache 和 L1 cache 之间必须是包含（Inclusive）的关系，当某个处理器核遇到 L1 cache 失效时，它首先产生一个总线请求（例如，BusRd 或 BusRdX），一旦总线访问请求被许可，该处理器核就可以访问 L2 cache 或者片上相邻处理器核的 cache 以完成 L1 失效的处理。在 CMP 中支持一致性与 SMP 中支持一致性没有什么区别，这在第 5 章中已经讨论过。SMP 系统的主要区别在于，某个数据块可能不在 L2 中，而共享 L2 cache 失效会触发对片外 L3 cache 或者内存的访问，此时，协议操作的请求者可能需要等待很多个时钟周期，因此 L1 cache 协议必须能够应对这种长延迟问题。

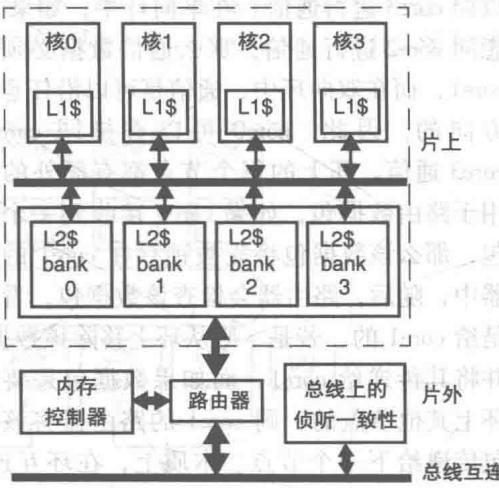


图 8-8 基于总线和共享 L2 cache 的四核 CMP 设计

在图 8-8 的结构中，共享 L2 cache 连接到外部路由器，该路由器再将存储器和其他外围设备（如磁盘）连接到片上，多个 CMP 可以通过外部总线将其互连。和单芯片上的多核互连不同，外部总线连接的是多个芯片，且每个芯片包含多个处理器核。

图 8-8 中的架构代表的是早期 CMP 架构，类似 Pentium 4 双核 CMP，在这种架构中，片上有两个 Pentium 4 处理器，且通过专用的片上总线接口互连。实际上，早期的 CMP 甚至都没有共享 cache，而是两个处理器核配备自己私有的 L2 cache。有意思的是，在这种 CMP 中，处理器核可以通过片上总线接口或片外前端总线接口物理上访问彼此的 L2 cache，因此，这里的双核就好像 SMP 系统中的两个处理器。简而言之，这种 CMP 架构就好像把一个双处理器 SMP 系统移到了片上。

基于环的 CMP

随着片上晶体管密度的不断增大，CMP 架构的性能也大幅度提升。图 8-9 给出了一个更先进的四核 CMP 框图，片上处理器核通过环网络进行互连，所有处理器核共享 L2 cache，核与

cache bank 之间也通过环网络通信。在总线结构下,任意周期中只有一个节点可以访问总线,与此不同,环网络允许多个节点在同一周期同时进行相互之间的通信,只要它们不会同时使用同一条链路就行。比如,在图 8-9 中,core0 可以同 core1 进行通信,与此同时,core2 也可以同 core3 进行通信。在单向环中,如果 core0 想同 core2 进行通信,那么通信数据必须经过 core1,而在双向环中,通信是可以沿任意一个方向的,因此,core0 可以直接同 core1 和 core3 通信。环上的每个节点都有额外的逻辑用于路由数据包,如果 core1 接收到一个数据包,那么该数据包将先被锁存在 core1 的路由器中,随后,路由器会检查该数据包,看是否是给 core1 的,若是,则从环上移除该数据包,并将其传递给 core1,而如果数据包是要发给环上其他节点的,则 core1 的路由器将该数据包传递给下一个节点。本质上,在环互连中,核间通信可能需要多跳才能到达最终的目标核,每次跳转通常需要一个周期,因此核间通信延迟会随着中间节点数目的变化而变化。一方面,环网络通过支持不同链路上的并行访问,提高了带宽和可扩展性;另一方面,基于环的互连网络在每个节点中都需要路由单元,这也增加了芯片的复杂度和面积开销。

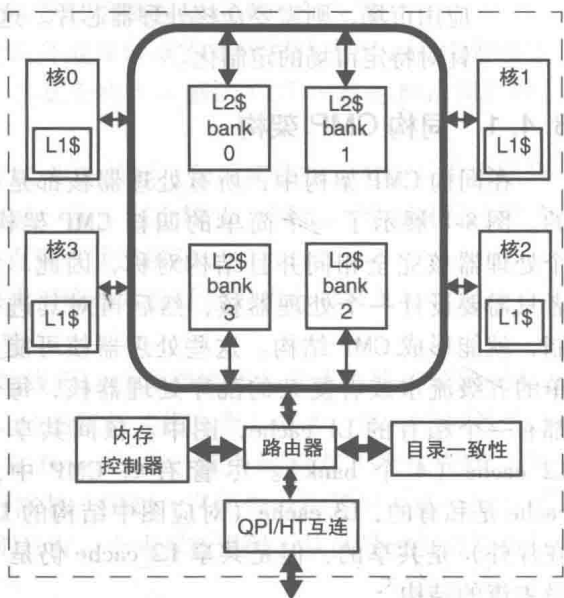


图 8-9 基于环互连的改进型四核 CMP

与基于总线的 CMP 结构相比,基于环的 CMP 需要更复杂的机制来支持 cache 一致性。由于并不是环上的每个节点都能看到所有的请求,如果要支持基于侦听的 cache 一致性,则必须将所有的一致性通信都使用显式的消息广播形式。例如,当 core0 产生一个 cache 一致性请求时,它必须显式地将信息发送到环上所有的处理器核上,该一致性请求从源节点出发,遍历整个环,每个节点接收到请求时,都检查自己的本地 cache 以确定所请求数据是否可用,如果可用,则更新一致性请求(但是不能将请求从环中删除),并继续传递给环上的下一个节点,一致性请求最终回到源节点,并携带了哪个节点包含最新数据的信息。一致性请求所需的延迟与请求节点和数据所在节点之间的相对位置无关,因此,环结构下侦听协议的行为类似统一内存访问(UMA)互连结构(比如共享总线)。

在环网络上也可以支持基于目录的 cache 一致性,这种结构下,环上的每个节点都作为一个特定地址段的主节点(home node),例如,core0 是地址 0~1K 段的主节点,core1 是地址 1K~2K 段的主节点,以此类推。主节点知道它对应地址范围内的数据块是否是脏的,并通过维护一组“存在”位(系统中的每个节点用一位表示)和一个“脏”位来跟踪有效副本的位置。所有的一致性请求先被发送到主节点,主节点再查找该数据块对应的目录项,并根据一致性要求做相应处理。当主节点不是拥有数据的节点时,该请求就被转发到脏数据所在节点,如果脏数据所在节点在请求节点和主节点之间,那么请求需要在环上多绕一圈。

图 8-9 代表了比较新的 CMP 结构,例如 Intel 的酷睿 i7 处理器,该处理器片上包含 4 个核,采用环互连网络,处理器核和 L2 cache bank 之间通过环网络进行通信,片外通信必须通过芯片的外部引脚来完成。由于引脚数目有限,CMP 的最大设计挑战之一就是片外带宽。为了尽可能减少片外的通信,酷睿 i7 配备了 L3 cache。除了大量的 cache,i7 还提供一套名为快速通道互连(QPI)架构的片外通信专用网络,QPI 使用点对点互连,传输带宽高达 25.6GB/s。

交叉开关互连

基于总线的和基于环的片上网络相对来讲都比较容易实现，但随着片上处理器核数的增加，这两种架构的扩展性都很差。例如，基于总线的互连结构带宽固定，无法随着核数增加而扩展。而要想在 CMP 系统中获得好的性能，高效和高带宽的核间通信是关键，尤其是片上核数很多时。图 8-10 给出了一个共享 L2 cache 的八核 CMP 结构，L2 cache 分为 8 个 bank，每个处理器核通过一个 8×9 的交叉开关连接到 L2 cache bank 和 I/O 上。在这种架构中，每个处理器核可以同时访问多个 cache bank，但多个核不能同时访问同一个 cache bank。相比总线和环结构，交叉开关互连结构在设计和验证上都更加复杂。

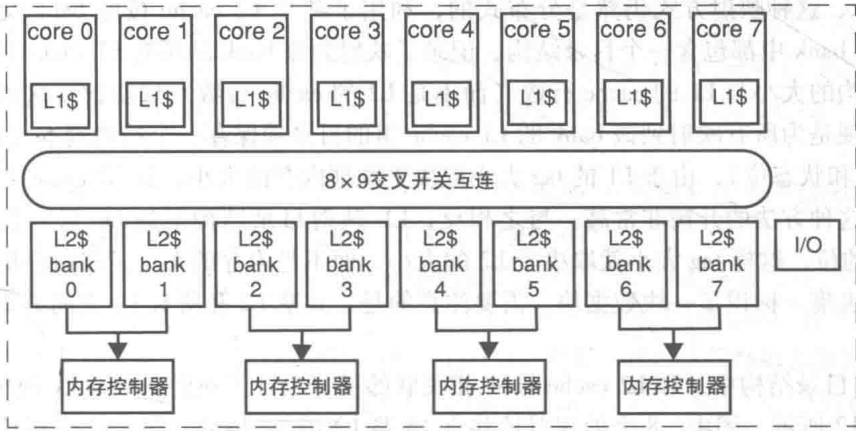


图 8-10 基于交叉开关的 CMP 结构：处理器核和 cache bank 及 4 个片上内存控制器通过 8×9 的交叉开关互连

共享 cache 架构

在复杂互连的 CMP 架构中，由于数据包路由具有动态性，因此，一致性常常通过点对点的目录协议来实现。目录必须包含 L1 cache 中块的全局状态的完整信息，在 MSI-无效协议的上下文中，该信息必须包括 L1 cache 副本的位置，以及指示数据块是否处于共享或修改状态。通常，L1 cache 是组相联的，且使用存储块地址的最低有效位进行组（set）索引，块地址的最低有效位也用来索引 L2 cache 的 bank 号，因此，使用相同位索引出来 L1 cache 中的所有的组都位于同一个 L2 cache bank 中，并且这些组内的所有一致性事务都在 cache bank 本地完成。如图 8-11 所示，L2 的 bank 选择位是 L1 cache 索引位的一部分，比如，假定 L2 cache bank（如图 8-10）数是 8 个，那么 L1 cache 索引的低 3 位指向 L2 cache bank，并且 L1 cache 组按照交错的方式映射到 8 个 L2 bank 中。

物理地址			
内存块地址			块内偏移
L1 tag		L1 cache索引	块内偏移
L2 tag	L2 cache bank索引	L2 bank号	块内偏移

图 8-11 共享 L2 cache bank 的 CMP 结构中，地址位到 L1 和 L2 cache 的映射方式

5.4 节介绍过一种名为“存在标志向量协议”的简单协议。假定共享 L2 cache 和私有 L1 cache 之间是包含关系，存在标志向量记录了 L1 cache 块的全局状态，并与 L2 bank 中的每个 cache 行相关联。例如，每个与 L2 cache 行相关联的目录项可能包含 n 个存在位（假定是 n 个处理器核）和一个状态位（共享或脏状态），就像基于目录的分布式共享内存系统中的做法。

当接收到 L1 访问请求，L2 bank 控制器就检查其对应 cache 组中的内容，当命中时查询对应的存在标志向量（该向量标记了 L1 cache 块的全局状态），基于查询到的状态，再按照类似于 cc-NUMA（见 5.4 节）系统中的方式同 L1 cache 控制器进行交互。和 cc-NUMA 中的主要区别在于：这里 L2 可能会失效，当出现这种情况时，将推迟一致性事务处理，且 L2 必须先从下一级存储中装载对应的数据块。

通常来讲，存在位向量会占用一定的存储空间，因为共享 L2 cache 的 cache 行数量通常比所有 L1 cache 行的总和还要多很多，这导致目录中有大量的目录项是用不到的。因此，另一种可能的目录组织方式是，在目录中维护 L2 cache 包含的所有 L1 cache 内容的映射关系，称为 L1-映射目录，这种映射方式仍然是分布式的，利用了共享 L2 cache 按照 bank 交错组织的特点。每个 L2 bank 中都包含一个目录结构，记录了映射到该 bank 的所有 L1 cache 的组内容，因此，目录结构的大小与 L1 的 cache 行数（而不是 L2 的 cache 行数）成正比。这种目录结构的一个简单实现是为所有映射到该 bank 的 L1 cache 组的目录项保存一个精确副本（包括 L1 的标识位（tag）和状态位），由于 L1 的 tag 大小取决于物理内存的大小，且 L2 cache 的大小远小于内存，因此这种方法的开销非常高。与之相反，L1-映射目录结构下的 tag 只应包含用于识别 L2 cache 行的位，这样 tag 大小就取决于 L2 的大小，而不是内存的大小。和内存地址一样，L2 中的块位置也唯一标识了一块数据块。需要注意的是，共享 L2 和所有 L1 之间需要维持一种包含的关系。

L1-映射目录结构中，和 L2 cache bank 相关联的部分包含了映射到该 bank 的所有 L1 cache 行，如图 8-12 所示。图中，8 个处理器核共享 16 路 L2 cache bank，每个处理器核配备一个两路组相联的 L1 cache（假定是指令/数据统一的 L1 cache）。图中的例子中，L1-映射目录中对应不同处理器核的三个指针指向 L2 的第 10 个 bank 中的某个 cache 行，这些指针在 L1-映射目录中是作为标志位使用的，位数很少，每个指针必须能够确定 L2 cache 的 bank 索引号以及 L2 cache 行的组相联号。参考图 8-11，L2 cache bank 索引中虚线右侧的位是 L1 cache 索引的一部分，因此，已经隐含在 L1-映射目录的组索引中，无需包含在 tag 标识位中。为了确定 L2 中的组索引号，L2 cache bank 索引中除去包含在 L1 cache 索引位中的那部分之外（也就是虚线左侧的部分）的位必须包含在 tag 中，在 tag 标识位中还需要添加 4 位来表示组相联号。通过将表示 L2 bank 索引的位和表示组相联号的位连接起来，就形成了 L2 cache bank 中包含该数据块的 cache 行的物理地址。L1-映射目录的每项中还需要两个状态位（有效位和脏位）以便跟踪记录 L1 cache 数据块的状态。

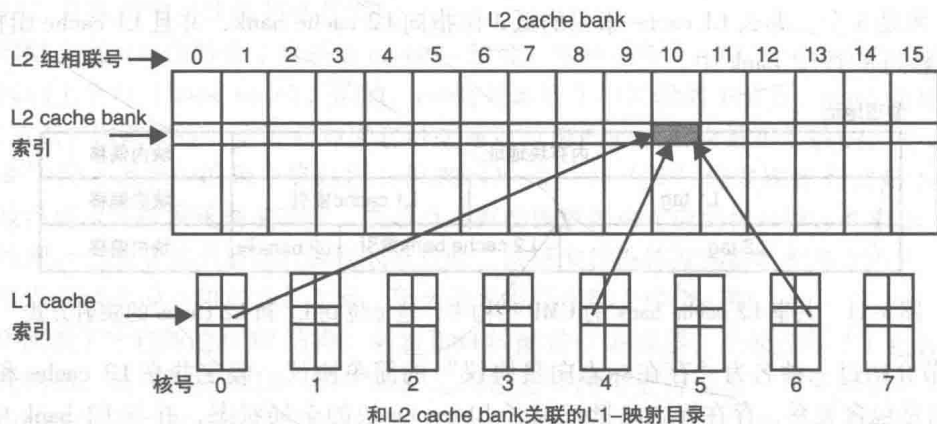


图 8-12 带 L1-映射目录的共享 L2 cache bank

当接收到来自 L1 cache 的请求时, L2 cache 的 bank 控制器先查找对应的组内容, 若命中, 则构建指向 cache 行的指针 (组相连号对应的位加上 L2 cache bank 索引位), 并将其作为访问 L1-映射目录的地址, 然后将来自目录的匹配响应组合成存在位向量, 然后协议就可以按照存在标志向量协议的方式继续处理了。

例 8.5 假定 CMP 系统包含 8 个处理器核和一个大小为 4MB 的共享 L2 cache, 每个处理器核包含一个 32KB 的 L1 指令 cache 和一个 32KB 的 L1 数据 cache, 两个 L1 cache 的组织方式都是四路组相联, 且 L1 cache 和 L2 cache 的块大小相等, 下面我们对存在标志向量目录和 L1-映射目录的位开销进行对比。

首先, L1 指令 cache 不需要维护一致性, 因此可以忽略。其次, 这里没有给出 L2 的 bank 数目, 但这并不影响我们进行对比, 因为我们要计算所有 bank 中 L1-映射目录的总和。此外, 这个问题也与 L2 中的组相联度无关 (也没有给出来), 并且, 这里也没有给出 cache 的块大小, 我们暂且用 B 来表示块字节数。

首先, 在存在标志向量目录情况下, 对应每个 L2 cache 行需要 9 位来进行标记 (每个处理器核 1 位, 再加上 1 个脏位), 因此总的位数为 $9 \times 4\text{MB}/B$ 。

在 L1-映射目录中, 系统中的每个 L1 cache 行都需要一个指向 L2 cache 行的指针, 以及 2 个状态位。指针的大小与 L2 中的 cache 行数目相关, 在这个例子中, L2 的 cache 行数为 $4\text{MB}/B$, 不过 L1 的索引位是隐含的, 因此从标志中删除。由于 L1 数据 cache 是四路组相联的, L1-映射目录的每项中包含 4 个标志。标志的大小为 $\log_2(4\text{MB}/B) - \log_2(32\text{KB}/4B) = 9$, 这样 L1-映射目录中每项的总位数为 11 (包括 2 个状态位), 因此所有 bank 的 L1-映射目录的总项数是所有 L1 cache 中行的总数, 在这里是 $8 (\text{核数}) \times 32\text{KB}/B = 256\text{KB}/B$ 。

因此, 存在标志向量目录和 L1-映射目录的位开销之比为 $(9 \times 4\text{MB}) / (11 \times 256\text{KB}) = 13$, 这个值与 L2 中的块大小、bank 数目、L2 中的组相联度等均无关, 它大概正比于 CMP 中共享 L2 cache 的大小除以所有 L1 cache 的总大小。

CMP 中的 cache 和内存带宽考虑

片外带宽是如今 CMP 系统中的关键参数, CMP 通过执行更多的线程来提高性能, 过去困扰体系结构发展的存储墙问题现已转化成存储带宽墙的问题。自 2000 年以来, 片外带宽已逐渐变得比内存访问延迟更加重要。片外带宽主要受引脚数量和频率的影响, 引脚数量限制了每个芯片上的访存通道数量, 当每个周期内的片外访存次数超过了内存通道数时, 就会有新的请求必须在芯片上排队, 以等待对内存通道的访问。这种排队会导致所有请求的访存延迟出现超线性增长。目前主要有两种处理片外带宽问题的基本方法: (1) 减少线程的带宽需求; (2) 增加可利用的片外带宽。

为了有效利用 CMP 上的多个处理器核, 应用程序必须是多线程的, 或者有多个独立的应用线程在不同的处理器核上并行执行。一般来说, 如果共享 cache 的访问速度足够快, 则共享 cache 总是优于私有 cache, 这是因为在相同尺寸下, 共享 cache 的有效容量通常要大于私有 cache 的有效容量。此外, 共享 cache 也没有一致性的问题。

当多个线程运行在同一个 CMP 系统上时, 共享 cache 可能出现两种效果: 要么相互合作共享, 要么相互干扰。在合作共享的情况下, 线程写入共享 cache 的存储块也可以被其他线程使用。例如, 当循环迭代在不同的处理器核上并行执行时, 所有核的指令都是相同的, 因此, 如果一个处理器核将循环迭代的指令取到共享 cache 中, 则其他所有处理器核都会受益。这也适用于共享数据, 当一个线程在共享 cache 中预取数据时, 它同时也是在为其他线程预取数据。由于共享的效果, 应用程序在合作共享场景下的片外带宽需求会有所降低。与之相反, 在相互

干扰的情况下,线程之间无法互相帮助,当多个独立的应用程序在同一个 CMP 上执行,且应用之间没有共享时,则应用程序很可能会将其他程序的数据从共享 cache 中替换出去,在这种情况下,每次在新的处理器核上启动一个新的应用程序时,都会增加其他正在运行的应用程序的带宽需求,因此应用程序的带宽需求会随着处理器核数的增加呈超线性增长。

以前,访存延迟是存储的主要问题,因此技术进步的作用显得非常关键,然而,在 CMP 系统中,情况却不一样了。比如,数据预取通常是在需要该数据之前,就将其从较低级别的存储层次中预取到 cache 中。预取是一种基于推测性的技术,它需要预测在不久的将来,哪些数据地址会被访问,并将对应的数据预取到 cache 中。当预测错误时,无效预取会由于线程读入了无用的 cache 块而导致片外带宽需求的增加。更糟糕的是,预取可能会因为替换掉了那些在不久的将来会再次被访问的存储块而污染 cache,并因此增加其他 cache 行的压力,cache 污染往往会进一步加剧片外带宽问题。预取准确性和 cache 污染甚至在单核处理器中都存在的问题,而在 CMP 系统中,问题就更明显。

随着集成水平的不断进步,芯片上将会集成更多层次的 cache 结构,且片上存储层次将会变得更加复杂,这一趋势早在 Power 7 处理器架构中就有所显现。在 Power 7 中,每个处理器核都有一个分离的私有指令/数据 L1 cache,和一个统一的私有 L2 cache,与此同时,片上还有一个大容量的共享 L3 cache (32MB),L3 cache 是采用内嵌 DRAM 技术实现的。在芯片上同 DRAM 集成在一起的还有随机逻辑和 SRAM 逻辑。

所有处理器核总的带宽需求迫使 CMP 系统必须通过高带宽的接口连接到片外资源,因此,CMP 是将系统组件集成到片上的主要驱动力。图 8-9 展示了如何将多个系统组件集成到片上以提升片外带宽。图中显示了片上集成的点对点互连接口,比如 Intel 的快速通道互连 (QPI) 和 AMD 的超传输 (HT) 接口,多个芯片间可以通过这种接口进行高带宽、低延迟的相互通信。除了芯片间的通信接口,目前的 CMP 还集成了片上存储控制器,以提供更高的内存带宽。例如图 8-9 中的芯片上也集成了内存控制器,该控制器运行在与主处理器相同的时钟频率下,相比于片外控制器的实现,集成在片上的内存控制器可以以更高的时钟速率执行访存协议,因此,可以显著降低访存延迟。Intel 的酷睿 i7 在片上内存控制器中支持了高速的 DDR3 存储接口,从而为所有处理器核提供了约 21 GB/s 的共享访存带宽。片上集成系统组件的不足之处是处理器被绑定在特定的接口标准上。例如,如果片上内存控制器实现了 DDR2 接口,则任何基于该 CMP 构建的系统都必须使用 DDR2 接口的双列直插存储模块 (DIMM),即使在不久的将来,DDR3 DIMM 可以提供更好的存储解决方案 (更高的带宽和更低的成本),也无法进行替换。

8.4.2 基于异构处理器核的 CMP 系统

除了系统集成能力的多样性,CMP 还提供了广泛的处理器核设计选择。考虑到处理器核的复杂性,在一种极端情况下,CMP 可能由多个简单的并行运行多个线程的顺序处理器核来构建,以此提高吞吐量性能;在另一种极端情况下,CMP 可能仅由少数几个规模大且能力强的乱序处理器核构建而成。大量的小处理器核非常适合于提高具有大量并行线程数的工作负载的吞吐量,比如商业或数据库工作负载;而少量的大处理器核则非常适合于提高单线程性能或线程并行度受限的工作负载的性能。

在许多应用情形下,运行在 CMP 上的应用程序会导致系统倾向于由异构组件构成。如果 CMP 中的处理器核数增多,且总的功耗固定不变,那么每个处理器核的功耗限制将超线性下降,这是由于片上网络 (如点对点网络) 的功耗会随着处理器核数的增多而超线性增长,这样留给处理器核的总功耗就会变得更少。因此,在含有大量简单处理器核的 CMP 系统中,依

赖于强大乱序处理器核的单线程性能将显著下降。除了少数用于科学计算的代码，大多数并行程序都有大量的串行代码穿插在并行代码中，在串行和并行代码中同时实现较好的性能是一个内在矛盾的任务。由于对微架构有截然不同的需求，在固定功耗限制下，同时优化单线程（串行部分）和吞吐量（并行部分）性能几乎是不可能的。针对单线程执行延迟的微架构技术有：乱序执行、推测执行和深度流水线等。在这些技术中，每条指令消耗的能量（EPI）相对较高，因此，在给定功耗限制下，CMP 中可容纳的 CPU 核数有限。另一方面，吞吐量性能的提升则要求大量的低功耗处理器核以充分利用线程并行性。

异构 CMP 很好地解决了单线程应用和多线程应用之间的需求矛盾。异构 CMP 中的处理器核可以在功能和性能方面均有所不同。性能的不对称可以通过多种方法实现，一种方法是基于总的并行度来调节总的 EPI，当应用只有少量并行性时，处理器可以将所有的功耗用于这些少数指令的处理；而当应用具有大量并行性时，处理器应当在每条指令上花费较少的功耗，以便支持大量指令的并行处理。具有不同 EPI 特征的异构系统可以通过多种方式来实现，比如，通过处理器核动态电压频率调节（DVFS）技术，或设计由不同功耗/性能特征处理器核构成的处理器，都可以很容易地改变 EPI，表 8-3 列出了四种实现异构特征的技术所对应的 EPI 范围。

表 8-3 实现异构处理器核的方法及其对应的 EPI 范围

方法	EPI 范围	改变 EPI 所需时间
DVFS	1:2 to 1:4	100μs, 调节 Vcc
改变可用资源	1:1 to 1:2	1μs, 填充 L1
推测执行控制	1:1 to 1:1.4	10ns, 刷流水线
不同特征核的混合	1:6 to 1:11	10μs, 迁移 L2

另一种改变 EPI 的方法是：根据应用的需求，动态改变处理器核的可用资源总量。例如，当运行工作集较小的应用时，可以关闭一些 cache bank 来减小 cache 大小和相联度，参考第 2 章的式 (2.12)，可知该方法可以有效地减少电容（C）和动态功耗，静态功耗也可以通过关闭 cache bank 的功耗而得以降低。此后，当应用的工作集又增大时，整个 cache 再重新激活。在很多处理器中，有 50% 的芯片面积都是分配给 cache 的，因此，通过及时关闭部分 cache，总的电容量最多可减少为原来的一半，EPI 也可以对应减少。同样我们也可以发现，即使在最理想的情况下，这种方法能节省的 EPI 也最多在 50% 左右。

我们还可以通过对推测执行进行控制（控制流水线中正在执行的指令数）的方式来改变 EPI。当分支预测器的准确率很低时，这种方法特别有用，因为无论错误推测执行的指令何时被清空，都会浪费大量的功耗。因此，我们不再激进地跨越多个未完成的分支指令继续取指，而是设定好一个阈值，当流水线中未完成分支指令数达到阈值时就阻塞流水线中的取指。通过对推测执行进行控制的方法可以获得的用于改变 EPI 的时间，大概和最坏情况下分支预测错误导致流水线清空的时间相等。由于限制取指只是影响了电路的活跃度，而大多数的处理器结构仍然在工作，因此通过控制推测执行的方法所能改变的 EPI 值比较有限。

上述 4 种实现异构的技术中，具体选择哪一种取决于应用中所能看到的 EPI 的变化次数，以及切换到新的 EPI 状态时的开销。比如，在 DVFS 技术中，如果应用程序的 EPI 需求比目前的电压/频率设定下的 EPI 更低，那么就可以继续减少处理器的电压和频率，不过，由于 PLL 同步的开销，改变电压和频率通常需要 10 ~ 100us 的时间。而要获得范围最宽的 EPI 调整，可以通过混合不同处理器核的方式来实现，其中一些是简单的顺序处理器核，而另一些是复杂的乱序处理器核（因为复杂乱序处理器核执行一条指令的功耗约为简单顺序处理器核功耗的 6 倍）。

除了性能差异之外，CMP 上的处理器核在功能上也是异构的。例如，某个处理器核可以为加密提供专门的支持，而另一个处理器核可以只运行图形密集的代码。在性能异构的 CMP 中，工作负载在不同处理器核之间移动，终端用户只能观察到执行时间的不同。比如，某个工作负载在复杂乱序处理器上运行只需要 10s，而在简单顺序处理器上运行则可能需要 100s。

相比之下，在功能异构 CMP 中，某些处理器核可能并不支持运行在其他处理器核上的代码执行所需的功能，因此，工作负载可能无法在处理器核间移动。为了使工作负载充分利用功能异构，必须提供在处理器核之间进行负载转移的机制，并对该机制清晰定义。这种机制在当前的浮点协处理器或图形处理器已经存在。下面我们探讨一下，这种机制在可以执行所有浮点操作的简单浮点协处理器上是如何工作的。当处理器核碰到浮点指令时，它将该指令的 PC 发送给浮点协处理器，并等待执行完成以及结果返回，该机制对软件层是透明的，完全由硬件进行处理。现在我们再考虑一个更加复杂的情形，处理器核需要将一个场景渲染的任务交给图形处理器进行处理，此时，主处理器提供一个应用程序编程接口（API）给软件开发人员，软件开发人员必须使用这些 API 来开发运行在图形处理器上的任务，当代码编译完成时，这些 API 被转换成专门的指令，当主处理器核在运行时，用这些指令来控制如何将任务转移到图形处理器上。

异构 CMP 的例子

图 8-13 所示的 IBM Cell 是 CMP 异构设计的一个极端例子，Cell 包含 1 个 PowerPC 处理单元（PPE）和 8 个协处理单元（SPE）。PPE 是一个能够处理复杂操作的两路 SMT 处理器核，而每个 SPE 是一个双发射的顺序处理器，每个周期至多可以发射 2 条 SIMD 指令。借助于支持 SIMD 的 SPE，Cell CMP 的向量处理能力有了显著提高。在 SPE 和 PPE 之间，没有用于支持存储一致性的硬件，因此，需要程序员来维护异构 CMP 上的一致性。

Intel 的酷睿 i7 通过 DVFS 对 EPI 的调节来支持简单版本的异构 CMP。Intel i7 通过名为“Turbo Boost”的技术来动态调整处理器核的工作频率，以此来匹配工作负载的变化和 CPU 的利用状况。如果 4 个处理器核中只有一个是活跃的，并且芯片的温度、功耗和电流是在额定范围内的，那么这个工作处理器核的时钟频率可以稍微提高（当前酷睿 i7 的实现方案中，频率可以提高 133MHz）。

8.4.3 连体处理器核

在连体处理器核构成的 CMP 中，处理器核之间共享资源的粒度比 cache 的共享粒度更细，连体处理器核可以共享如 ALU 这类的资源。连体处理器核是受到单个处理器内资源利用具有明显的时间差异性这一现象的启发而提出的，特别是复杂的资源，如浮点功能单元占据了大量的芯片面积，但对大多数工作负载来说，却并未得到充分利用。连体处理器核能够在片上核间共享这些占用大量面积但是利用率又很低的资源，这种细粒度共享是通过处理器核和共享资源间的高速互连来实现的。

为了有效地共享资源，在 CMP 中对处理器核进行布局时，必须考虑处理器核共享资源的访问模式。如果浮点单元（FPU）在同构 CMP 的多个处理器核之间共享，则在芯片布局上，

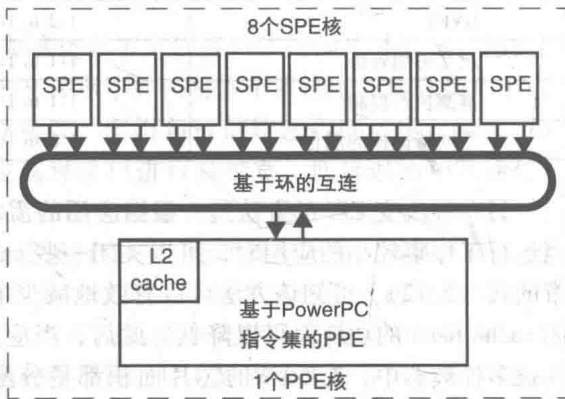


图 8-13 IBM Cell 异构 CMP

应该让每个处理器核的执行流水级与 FPU 保持等距, 这样所有处理器核对 FPU 的访问延迟才会相等。除了确保访问延迟, 在资源共享时, 还需要引入仲裁机制以防止多个处理器核在同一周期内都对共享资源进行访问。现有的、针对处理器核间共享总线的仲裁方法也同样适用于这里的资源共享的情况。需要注意的是, 连体处理器核和处理器核异构的概念在设计空间上是正交的, 换句话说, 人们既可以设计同构的连体处理器核 CMP, 也可以设计异构的连体处理器核 CMP。

图 8-14 展示了一个基于连体处理器核的 CMP 系统的可能实现, 该系统包含两个处理器核, 核间共享同一个 FPU。两条流水线的执行阶段被连接到一个请求缓冲区, 当执行阶段遇到浮点指令时, 该操作的两个源操作数被放置到请求缓冲区中, 仲裁逻辑决定来自两个处理器核的请求的服务顺序。需要注意的是, 来自同一个处理器核的多个请求会按照请求缓冲区接收请求的顺序来依次服务, 仲裁逻辑只决定来自不同处理器核的请求之间的顺序。

Sun 公司的 SPARC T1 处理器是基于连体处理器核的 CMP 结构的典型例子: T1 在每个芯片上有 8 个处理器核, 核间共享一个 FPU 以减少芯片面积开销。每个处理器核都支持细粒度多线程, 执行过程中遇到任

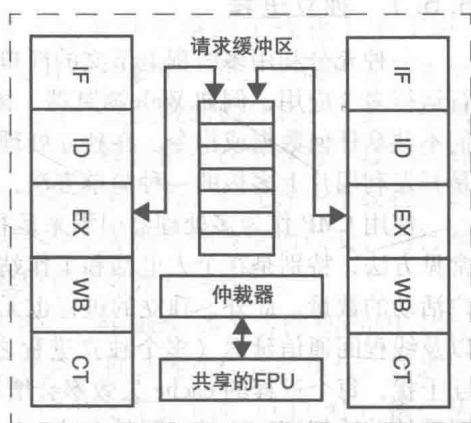


图 8-14 共享浮点单元 (FPU) 的连体处理器核

何浮点运算指令时, 就会将对应的核 ID、操作码、操作数在内的信息发送给 FPU 共享单元, FPU 计算结果, 并将其返回给请求处理器核。在这期间, 请求处理器核会暂停当前线程的执行以等待浮点运算完成, 同时会从其他正在运行的线程中取指执行。

8.5 编程模型

在 8.3 节和 8.4 节中, 我们介绍了各种不同的硬件多线程处理器核和 CMP 设计选择, 与此同时, CMP 系统对应的编程模型也有很多选择。片上多线程支持无需系统干预的快速上下文切换, CMP 具有低延迟的片上网络和共享的最后一级 cache (LLC), 这有助于提升核间通信的效率。上述硬件特性对软件开发人员的主要影响是: 开发人员可以创建出通信粒度更细、长度更短的线程。

例 8.6 线程通信开销的影响 为了说明在不同线程粒度下通信延迟的影响, 首先考虑一个简单的例子。应用开发人员需要判断一个循环的两次连续迭代 (称为迭代 1 和迭代 2) 是否可以在两个不同的线程上下文中并行执行, 假定迭代 1 需要花费 T_{ns} 完成, 并在 $T/2$ 时刻产生供迭代 2 使用的数据, 迭代 2 先运行 $T/2_{ns}$, 然后它必须在收到来自迭代 1 的数据后才能继续执行。假定在 SMP 系统中, 数据通信的时间为 100_{ns} , 在 CMP 系统中, 处理器核之间的通信延迟为 10_{ns} , 而在多线程处理器核中, 通信延迟为 0。针对上面每一种情况, 分析在 T 取什么值的情况下, 并行执行这两个线程会有优势?

首先考虑 SMP 的情况。迭代 2 需要 $T/2 + 100 + T/2_{ns}$ 才能完成, 由于迭代 1 和迭代 2 并行执行, 在 SMP 上的总执行时间是 $T + 100_{ns}$, 如果两次迭代在单个处理器核上串行执行, 总的执行时间将是 $2T$, 假定迭代 1 通过寄存器将数据传递给迭代 2, 没有通信开销, 为了使 SMP 上的并行执行更优, $2T$ 必须大于 $T + 100$, 即 $T > 100_{ns}$ 。

再考虑 CMP 上两个处理器核运行相同迭代的情况。当 $2T > T + 10$, 即 $T > 10_{ns}$ 时, CMP 上

的并行执行效果会更优。因此，在 CMP 上，为了利用并行性，每个线程的大小只需是 SMP 上线程大小的十分之一。

最后，考虑两个线程运行在同一处理器核上的情况。在这种情况下，从通信零延迟的角度来看，并行执行结果总是更优的。

本节中，我们主要探讨从软件的角度如何利用硬件架构提供的各种特征。

8.5.1 独立进程

一种充分利用多线程上下文的简单方法是运行多个独立的进程。例如，台式机用户通常并行运行多个应用，例如 Web 浏览器、流媒体视频和文档编辑器，这些应用之间相互独立，它们不共享任何数据或指令。在独立处理器核或线程上下文中，以独立进程的方式运行每个应用是开发利用片上多核的一种简单方法，不需要用户或者应用开发人员做什么工作。

使用 CMP 作为多处理器引擎来运行独立进程，这是开发利用处理器核和线程上下文的最常见方法，特别是在个人电脑和工作站的商品市场中，不过这种方式下并行的数量会受限于用户活动的数量。此外，独立的进程也无法利用 CMP 的共享资源（多个独立进程是互相竞争的）以及线程间通信延迟（多个独立进程之间不传递数据）更短的特征。比如，由于进程间的相互干扰，每个进程的 cache 失效率会增加，访问片外存储的总线请求率也会提高，从而导致更严重的总线拥塞和更长时间的 cache 失效开销。总之，这种方式下不能利用共享 L2 cache 所带来的任何好处。

8.5.2 显式线程并行

多进程处理的方式依赖于各个用户处理多任务的能力，此外，也可以用于多个用户共享同一台机器的场景。虽然多进程方式提高了 CMP 的资源利用率，但每个用户任务的执行时间却保持不变，甚至可能由于资源共享的负面影响而变得更加糟糕。到目前为止，单线程的性能大多通过开发指令级并行（ILP）以及缩短时钟周期时间来实现，随着 ILP 提升的难度越来越大，线程级并行（TLP）成为提升应用性能的新方法。在这种方法中，单个应用被划分成多个线程分配到多个线程上下文中并行运行，线程之间通过紧密交互来完成数据交换。从应用中提取 TLP 来加快执行已成为 CMP 系统的关键。事实上，除非应用并行化的目标可以通过某种方式解决，否则通用架构在未来是无法充分利用摩尔定律带来的资源优势的。不幸的是，软件并行化是很困难的一件事，并且由于用户的抵触，一些遗留软件甚至无法进行并行化，此外，很多关键的软件（如编译器）也很难并行化。

程序员需要仔细理解应用程序的语义，识别出那些可并行执行的代码片段，以此开发出多线程代码。循环迭代和函数调用是两种常见的语言结构，程序员很容易从中找到可并行的线程，只要循环依赖关系可以消除或者处理掉，不同的迭代就可以并行执行。有效地识别循环依赖关系，尽可能编写出高效通信的代码是程序员的责任。循环中的并行迭代对任何共享数据的修改必须通过互斥锁进行保护。一个共享数据变量的例子是循环迭代计数，由于每个线程只在每次循环迭代的结尾对计数器进行递增，因此计数器的递增过程必须在互斥锁的保护下串行完成。当每个线程完成分配给它的循环迭代次数时，可能还需要等待其他所有线程都执行完才能继续推进，这种等待可以通过线程同步原语来实现。应用开发人员需要在应用中插入合适的线程同步代码，硬件只是为互斥锁和 barrier 同步提供简单的原语支持。

程序员可以借助软件库来管理并行线程的执行，这样可以简化线程的创建、同步、通信、退出等。线程库 API 提供了对应的宏或者函数调用用于线程管理，它们将程序员与具体底层硬件实现的同步指令进行分离。Pthread 是一个流行的线程库例子，它为程序员提供了并行的

API，通过该线程库，程序员可以创建并行执行的线程，在互斥锁的保护下访问共享数据，并调用同步函数。

基于 OpenMP 的编程 API 以标记的形式提供了编译制导指令，以帮助应用开发人员识别出哪些代码段适合并行化的，程序员通过在循环的开始处插入 OpenMP 编译制导语句来标记循环是可并行化的，这些编译制导通过 OpenMP 编译器和针对每个硬件平台的运行时库相结合，自动转换成并行线程。OpenMP 编译器和运行时库按照编译制导的要求将代码翻译成尽可能多的并行线程，并将一组循环迭代分配给线程上下文。当底层硬件支持两个线程上下文时，OpenMP 运行库就会产生 2 个线程，每个线程并行执行总循环迭代数的一半。当硬件支持 4 个线程上下文时，就自动产生 4 个线程，每个线程并行执行总循环迭代数的四分之一。循环结束时，OpenMP 运行时环境会自动插入一个 barrier 同步原语，每个线程都在这里等待，直到所有线程都完成各自的循环迭代。barrier 同步后，运行时环境挂起所有线程，只允许一个线程串行地执行循环后的剩余代码。在应用程序中指定的编译制导指令不随硬件的改变而改变，运行时环境知道底层硬件结构，并会自动创建足够的并行线程数以更高效地利用硬件。

表 8-4 给出了一个基于 OpenMP 并行的简单例子。在表中，#pragma 指令告诉 OpenMP 运行时环境，该指令下面的循环没有循环依赖关系，且每次迭代可以同其他迭代并行执行。如表中所示，如果代码执行在拥有两个线程上下文的系统上，第一个线程上下文（CPU#1）执行从 0 ~ ($n/2 - 1$) 个迭代，第二个线程上下文执行循环迭代的下一半；如果代码执行在拥有 4 个线程上下文的系统上，运行时环境自动分配四分之一的循环给每个 CPU。

表 8-4 基于 OpenMP 制导指令的自动并行

#pragma OpenMP parallel for(int i=0; i<n; ++i(a[i] = b[i] × c[i])				
CPU#	1	2	3	4
在 2 个线程上下文中运行	$i = 0 \text{ to } (n/2 - 1)$	$i = n/2 \text{ to } n$		
在 4 个线程上下文中运行	$i = 0 \text{ to } (n/4 - 1)$	$i = n/4 \text{ to } (n/2 - 1)$	$i = n/2 \text{ to } (3n/4 - 1)$	$i = 3n/4 \text{ to } n$

OpenMP 库和运行时环境是可以分别有不同实现的，下面描述一个简单的方法来说明 OpenMP 库和运行时是如何配合实现并行执行和跨线程同步的。程序中的并行循环标记被简单翻译成 OpenMP 的库函数调用，以实现循环的并行执行，库函数在运行时会先检测底层系统，以获得该循环可用的线程上下文数，之后，库函数尽可能地创建与硬件线程上下文数相同的线程。

执行循环的每个线程可以看成是一个函数调用，循环索引的起始和结束作为函数调用的输入参数，循环索引的起始和结束值可以根据线程上下文数来确定。然后，运行时环境给每个线程提供了相同起始地址的函数调用，但是传递不同的开始和结束循环索引作为函数调用参数。当线程完成分配给它的任务时，就返回到生成该线程的 OpenMP 库调用，库函数中的 barrier 同步原语会阻止任何线程执行循环后的指令。当所有线程都到达 barrier 同步点时，运行时环境会挂起所有线程，只保留一个线程继续执行并行循环后面的指令。

OpenMP 提供了大量的制导指令用于声明一些在并行线程创建前需要在运行时环境中检测的条件。比如，上面的例子中如果 n 的值太小，循环并行化可能并不明智，因此用户可以指定 n 的最小值，只有 n 的值比用户指定的值大时，并行才会生效，这项检查在运行时进行。提供大量 OpenMP 制导指令的目的是将程序员的负担转移到运行时环境，实现线程的自动创建并正确同步。尽管如此，OpenMP 仍然依赖于程序员的经验知识，来指定代码中的并行机会以及并行相关的各种限制。

8.5.3 事务内存

OpenMP、Pthread 以及其他并行编程模型可以帮助应用开发人员快速编写出并行代码，然而，在这些显式的线程并行方法中，并行代码开发的主要瓶颈仍然是，需要依赖程序员来确定并行代码段，并使用正确的同步原语来管理线程间通信。在代码并行化中，确定并行代码的行为并决定何时使用同步和互斥锁是最为关键的任务。当锁和临界区出现嵌套时，还可能发生死锁。

无论是由编译器自动产生的，还是由程序员手写生成的，并行代码的性能都依赖于消除不必要的线程同步。当多个线程想要更新一个共享数据结构时，通常需要使用锁或 barrier 来对数据访问进行同步。两个不同线程间的数据结构共享有两种类型：真共享和假共享。真共享发生在两个线程修改/读取数据结构的相同字段；假共享发生在两个线程访问相同的数据结构，但修改/读取的是数据结构的不同字段。如果两个线程总是访问数据结构的相同字段，那么就没有必要对其进行串行化处理，因此保护数据结构的锁或 barrier 可以删除。然而，实际困难在于，如何静态（在程序编写或编译时）确定共享的数据类型，特别是当线程间共享数据会动态变化时就更加困难了。我们也可以对细粒度的共享数据结构进行加锁，而不是对整个大的共享数据结构加锁，但细粒度锁会导致锁的数量增多，很难跟踪记录和管理，并增加死锁发生的几率。除了假共享和死锁问题，在线程可以被中断和抢占的环境下，基于锁的同步还会引发其他严重的问题。如果某个线程持有锁或者是 barrier 同步协议的一部分，并且被抢占（比如，由于缺页异常）调度出去，则所有参与了相同计算的其他线程也都会被锁定，并可能面临极其长时间的延迟。

锁定整个数据结构是确保线程正确通信的一种极端方法。比如，有一个上千节点的图结构，可以通过若干线程进行并行搜索和修改，修改可能是对某个节点上下文的更新，或者是删除/添加节点。在这种情况下，任何节点都可能被两个线程共享读/写，但同时发生的几率又非常小。不过为了确保不会出现竞争情况，程序员必须锁住整个图、一部分图或者是图中的每一个节点，然而在大多数时间里，这种串行化和加锁行为都是没必要的。

例 8.7 基于锁的动态数据结构共享 图 8-15 所示的简单代码片段说明了使用锁来优化线程同步所面临的困难。在这个例子中，生产者和消费者线程之间的共享数据随着 cond1 和 cond2 的变化而产生变化，如果 cond1 和 cond2 都为真，或者都为假，则两个线程修改同一个数据项，此时临界区是必要的。然而，如果 cond1 为真，cond2 为假（反之亦然），则两个线程更新的是不同的数据项，此时临界区就没有必要。由于 cond1 和 cond2 的值在编译时无法确定，并可能动态改变，程序员必须决定最佳的加锁粒度，因此，程序员可能倾向于在临界区内执行生产者和消费者的功能，从而避免在细粒度共享情况下使程序分析和调试更加复杂。图 8-15a 中就是这么做的，这种方式本质上是在所有情况下把代码串行化了。如果 cond1 和 cond2 都为真或都为假的情况很少发生，那么这种串行化在大部分时间里是没有必要的，也导致了大量的 TLP 被浪费掉。

基于事务的编程

事务内存（Transactional Memory, TM）也是一种并行编程模型，它可以消除程序员选择最佳锁粒度的负担，也可以消除加锁和执行互斥的开销。事务内存通过支持无锁数据结构来增强并行程序的可编程性和性能，这是一个乐观的实现方法，它假定共享对象的数据项很少会出现并行读写，因此代码对它们的访问可以在不锁定的情况下执行。在执行过程中，当检测到同时读/写相同数据结构的相同数据项这样的罕见事件发生时（即检测到竞争条件），某些执行过


```

struct Shared{
    int item1;
    int item2;
} myShared;
Producer(){
    Lock(1);
    if(cond1)
        myShared.item1++;
    else
        myShared.item2++;
    Unlock(1);
}
Consumer(){
    Lock(1);
    if((cond2)&&(myShared.item1>0))
        myShared.item1--;
    if((!cond2)&&(myShared.item2>0))
        myShared.item2--;
    Unlock(1);
}

```

a)

```

struct Shared{
    int item1;
    int item2;
} myShared;
Producer(){
    Transaction_Begin
    if(cond1)
        myShared.item1++;
    else
        myShared.item2++;
    Transaction_End
}
Consumer(){
    Transaction_Begin
    if((cond2)&&(myShared.item1>0))
        myShared.item1--;
    if((!cond2)&&(myShared.item2>0))
        myShared.item2--;
    Transaction_End
}

```

b)

图 8-15 基于锁 (a) 或事务 (b) 的数据结构动态共享的例子

程就会进行回滚。基于 TM 的一个简单实现中，应用程序可以像常规代码一样来实现复杂的同步协议，它只需要将协议代码包含在一个事务里，而无需显式的同步。与临界区机制相反，不同的事务之间只要不存在读/写竞争就可以并发执行，对读/写竞争的检测是由硬件自动完成的，如果动态检测到竞争，那么同步相关的代码将自动重新执行。

当事务回滚的可能性很低时，事务内存机制就非常高效。前文中描述的多个线程对大规模图的搜索和修改就是一个很好的例子。相比之下，数值应用中通常使用 barrier 进行同步（参见第 7 章中的图 7-2），这种应用出现回滚的概率很高，因此，对事务内存机制来说，数值应用可能并不是一个好的选择。相比于基于锁的基本共享内存编程方法，TM 具有更好的编程性，这是因为并行程序员只需要确定每个线程中的事务边界，而不需要去考虑复杂锁方案的正确性。在 TM 编程中，主要关注的是，需要尽可能创建比较小的事务，从而最小化回滚的可能性以及每次回滚时需要重新执行的次数。当然，也可以将整个线程放在一个事务中，这在实现起来非常简单，但是回滚线程整个执行过程的可能性也会非常高。TM 的另一个问题是，不能进行回滚的动作（如对物理设备的访问）不能包含在事务中。因此，任何 I/O 操作都不能包含在事务中。通常情况下，线程产生的效果中，唯一可以进行回滚的就是对内存的修改。

TM 的概念是从数据库事务中借鉴而来的。在数据库中，事务是一系列作为原子块执行的指令，也就是说，事务中所有指令以原子块的形式进行提交或中止，事务不能拆开执行，在任何情况下事务未能成功完成时，都必须重新执行。除了原子性，事务的第二个必要属性是隔离性，隔离性可以防止其他事务在该事务提交之前就观察到其行为，隔离性常被理解为原子性的一部分。最后，事务必须是可串行化的，即已提交事务的结果必须按线程提交顺序对其他线程可见。

原子性意味着事务可以对共享内存做一些读/写操作，并且所修改的值对其他事务暂时不可见，当事务提交时，存储上的更新对其他所有线程都变得可见，就好像所有的这些修改是立即完成的。如果事务中止，则所有改变对任何其他线程都不可见，如果线程在事务处理中被抢占，首先会中止该事务，从而使其他线程可以访问到共享变量。

例 8.8 基于事务的动态数据结构共享 图 8-15b 中的代码与图 8-15a 中的代码功能相同, 但是用事务机制代替了锁。transaction_begin 语句表示事务开始, 它意味着本地处理器必须开始监控对存储的访问, 直到碰到 transaction_end, 再对事务的隔离性进行验证。如果没有违反隔离性, 则事务在 transaction_end 处提交, 并且事务内的 store 内容对所有线程可见, 而如果违反了隔离性, 则事务必须中止、回滚, 并重新执行。当两个线程的大部分时间都用于更新共享变量 myShared 的不同数据项时, 事务代码比传统基于锁的代码更高效, 这是因为事务代码无需串行化, 无需执行全局同步 (acquire 或 release), 并且线程很少对事务进行回滚。 ◀

事务内存机制

除了回滚和重启事务的能力, 在所有 TM 架构中, 还必须具备三个基本机制来对原子性和隔离性进行检查和维护。

- 事务冲突检测机制。当两个或更多事务访问相同的存储单元, 且至少有一个访问是 store 操作时 (如果只有 load 操作则无需考虑), 将会检测到事务之间的冲突, 这违反了隔离性。
- 推测性内存数据管理机制。未提交事务中的 store 推测值必须与已提交的 store 值 (来自已提交事务) 分开, 事务内存系统的这一功能通常称为版本管理。
- 并发控制机制。当检测到事务冲突时, 该机制用于决定哪些事务需要中止, 哪些事务可以提交。

从开始到提交, 事务必须在本地记录其读集合 (即已加载的地址) 和写集合 (即已修改的地址), 以此来检测事务冲突。为了确保事务的隔离性: (1) 其他事务不能对写集合中的地址进行读或写操作; (2) 其他事务不能对读集合中的地址进行写操作, 否则, 必须中止其中的一个事务。

通常情况下, 冲突检测和版本管理可能包含两种实现方式: 积极方式或懒惰方式。如果事务冲突一发生就能检测到, 则这种冲突检测是积极的; 如果在事务冲突发生之后才被检测到, 则是懒惰的, 最晚的可能检测时间是事务提交的时间。如果在事务开始时, 初始的有效值先被保存在其他位置的内存中, 而新产生、非提交的值被直接存储到内存中, 则是积极的版本管理方式; 如果非提交的值被先存储在缓冲区中, 事务提交时, 才被写入内存中, 则是懒惰的版本管理方式。一般情况下, 积极的冲突检测和版本更新策略更有优势。通过积极的冲突检测, 事务中止和回滚可以尽早执行, 从而避免更多无用的工作。通过积极的版本更新, 可以加快提交速度 (因为所有新值早已在内存中, 而保存了初始值的缓冲区只需要简单清空即可), 但是中止速度会变慢 (因为内存值必须从缓冲区中进行恢复)。懒惰的冲突检测和版本管理方式的效果则正好相反。通常情况下, 我们希望提交比中止更加频繁, 因此积极的版本管理方式通常更为理想。

按照 TM 机制的实现方式方式, TM 系统可以分为硬件事务内存 (HTM) 和软件事务内存 (STM), 而混合事务内存系统则同时利用了硬件和软件来实现事务内存的目标, 下面主要讲述硬件事务内存。

事务内存系统的基本硬件结构

为了简单起见, 不考虑嵌套事务, 只专注于事务中那些访问共享内存的地址集合。通常情况下, 事务外的访存地址采用与非事务性系统相同的处理方式。

我们在 ISA 中增加两条新指令来开始和结束事务, 这两个指令的操作码分别为 TBegin 和 TEnd。TBegin 指令执行的效果是将处理器切换到事务状态, 当执行到 TEnd 指令时, 处理器又切换回非事务状态, 这两条指令对处理器中的状态位进行置位和重置, 从而记录处理器是否正

在执行事务，该状态位称为 `transaction_active` 位。

当事务开始时，正在运行的线程状态必须设置好检查点 (checkpoint)，此时记录的状态包括寄存器和内存。由于寄存器的值是每个线程私有的，因此检查点的实现比较简单。例如，可以设置一个影子寄存器文件来保存寄存器的检查点，当 `TBegin` 执行时，影子寄存器文件中的所有项都是无效的。在事务中，新的寄存器值被存储到影子寄存器文件中，读寄存器时先在影子寄存器中查找寄存器的值，如果影子寄存器中对应项是无效的，则从主寄存器文件中取值，如果事务在 `TEnd` 处成功提交，则影子寄存器中的有效值被复制到常规的主寄存器文件中。

和寄存器值的保存不同，如果要对线程的内存状态进行检查点保存，那么难度就大得多，这是因为内存值可能被其他线程共享。下面给出一个简单的基于事务 `cache` 状态的机制，简而言之，当事务中的内存块第一次更新时，内存系统中必须维护该数据块的两个副本：在第一次 `store` 执行之前的数据块副本（已经提交的值），以及新的包含未提交 `store` 值的数据块副本。处理器核的 `L1 cache` 可以保存新的未提交数据块副本，下一级存储（共享 `L2 cache` 或者内存）可以保存在事务开始时有效且已提交数据块副本。下面的 `cache` 协议是一个简单的基于总线侦听的 `MSI-无效` 协议，该协议中存储块状态包含如下几个状态：`Modified`（单副本，内存值是旧的，支持读/写访问），`Shared`（一个或多个副本，内存值是最新的，支持只读访问），`Invalid`（数据不在 `cache` 中）。有关该协议的更多细节请参见第 7 章。

`L1 cache` 中的数据块可能处于两种状态：正常状态或事务状态。正常数据块包含已提交的值，而事务数据块包含的是数据块的新版本。如果当前事务成功提交，则新版本数据会代替旧的已提交值，如果当前事务被中止，则新版本数据块将被丢弃。正常块和事务块的主要区别是，事务块中的值在事务提交之前对其他线程不可见。任何时候，同一内存块只能有一个副本保存在 `cache` 中。

在事务机制中，所有的 `load` 和 `store` 都被视为特殊的 `TM` 访问指令。事务中第一次对内存块进行 `store` 操作时，该内存块的当前副本必须进行检查点保存，并创建一个新的事务副本，此时需考虑以下两种情况：

- 该内存块不在 `L1 cache` 中。在这种情况下，`cache` 必须首先获取内存块的一个 `Modified` 的副本，且对内存或 `L2 cache` 进行更新。然后，在事务状态下，该 `Modified` 副本装载到 `cache` 中，共享 `L2` 或内存中的副本作为检查点副本。
- 该内存块在 `L1 cache` 中。如果是 `Modified` 的副本，必须首先将它写回（检查点副本）；如果是 `Shared` 的副本，则不需要写回，因为下一级存储中的副本也是最新的。然后，在事务状态下，获取 `Modified` 的副本并加载到 `L1 cache` 中。

在事务执行过程中，第一次 `store` 之后的所有其他 `store` 都会对本地 `L1 cache` 中的事务副本进行更新。在该事务的剩余执行过程中，该内存块必须一直在 `cache` 中并处于事务状态。当事务中执行一个对普通内存块的 `load` 操作时，`L1 cache` 中对应的 `cache` 行会标记为 `read`（通过设置和每个 `cache` 行关联的 `transaction_read` 位），此后该内存块必须始终保留在 `cache` 中，直到事务结束。

事务中的写集合通过事务块副本来维护，事务中的读集合则通过 `transaction_read` 位来维护，这些都在本地 `L1 cache` 中实现。冲突检测使用事务状态位和 `transaction_read` 位，多个线程间的冲突检测通过 `cache` 一致性协议来实现，为了支持 `TM`，`MSI-无效` 协议在以下几个方面进行了修改：对正常内存块或事务外部处理器的 `cache` 发起的总线请求按常规的 `MSI-无效` 协议进行处理；当处理器在事务内部时（通过 `transaction_active` 位标记），并且收到一个针对事务块的 `BusRd` 或 `BusRdX` 请求时，则检测到冲突；此外，如果收到的请求是 `BusRdX`，`cache` 副本块是正常状态且 `transaction_read` 位为 1，则也说明检测到冲突。当检测到冲突时，`cache` 向请求者发

送一个 busy 响应, 该响应用来确保事务的隔离性。当发出请求的处理器收到 busy 响应时, 必须过段时间再重新发起请求, 如果发起请求的线程自身也正在执行事务, 则比较明智的选择是中止当前事务并回滚, 而不是继续忙等, 从而避免事务间出现死锁。如果发起请求的线程在事务外部, 则该线程会被挂起, 直到远程事务提交之后, 再进行重试。

一旦事务被中止或提交, 所有 L1 cache 行中的 transaction_active 位和 transaction_read 位都将被复位。如果事务中止, L1 cache 中的所有事务项都将被丢弃 (通过快速重置相应 Valid 位), 此外, 影子寄存器的内容也将丢弃。另一方面, 如果事务成功提交, 则影子寄存器的内容将转移到主寄存器, L1 cache 中的所有事务块也将切换到正常状态 (通过快速重置所有的事务位)。需要特别注意的是, 当事务提交时, 整个提交操作必须满足原子性, 也即整个状态切换需要原子完成 (即快速复位 cache 中所有 cache 行的事务位, 或当事务块顺序切换至正常状态时, 不对其他的任何一致性请求做应答)。要做到这一点, 整个事务中, cache 中的所有事务块必须保持在 Modified 状态, 这样事务提交操作仍旧是本地操作, 不需要与全局存储系统进行交互。

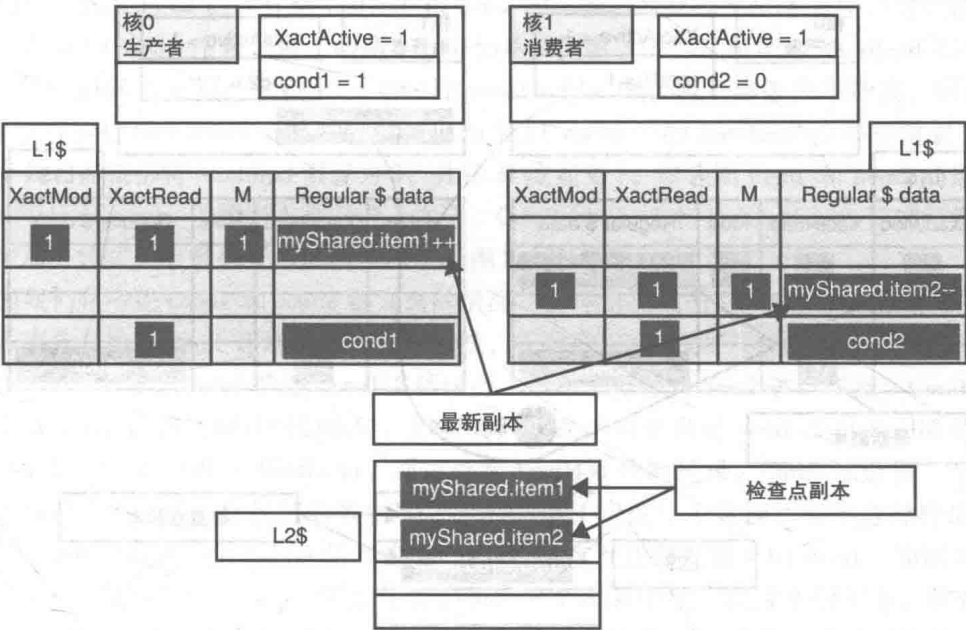
事务 cache

当前, 我们常常会忽视由于 cache 容量限制和 cache 映射冲突导致的 cache 替换, 因此, 随着事务中 store 数量的不断增加, 在某些 cache 组 (set) 中最终会出现溢出。当 cache 中某一组的所有 cache 行都被事务块或设置了 transaction_read 位的正常块占用时, 就发生了溢出, 此时对应 cache 组中会发生失效。为了缓解这一问题, 事务块可以分配到独立于主 cache 的全相联 cache 中, 这种方法将事务块的维护与主 cache 的管理分离开来, 可以并行地访问主 cache 和事务 cache。这两个 cache 是互斥的, 因此内存块副本不能同时存在于这两个 cache 中。当事务提交时, 事务 cache 中的所有块都变为正常块, 因此, 事务 cache 中包含无效块、正常块和事务块, 当需要分配新的事务块时, 事务 cache 中的无效块或正常块被替换出去, 以便为新的事务块提供空间。

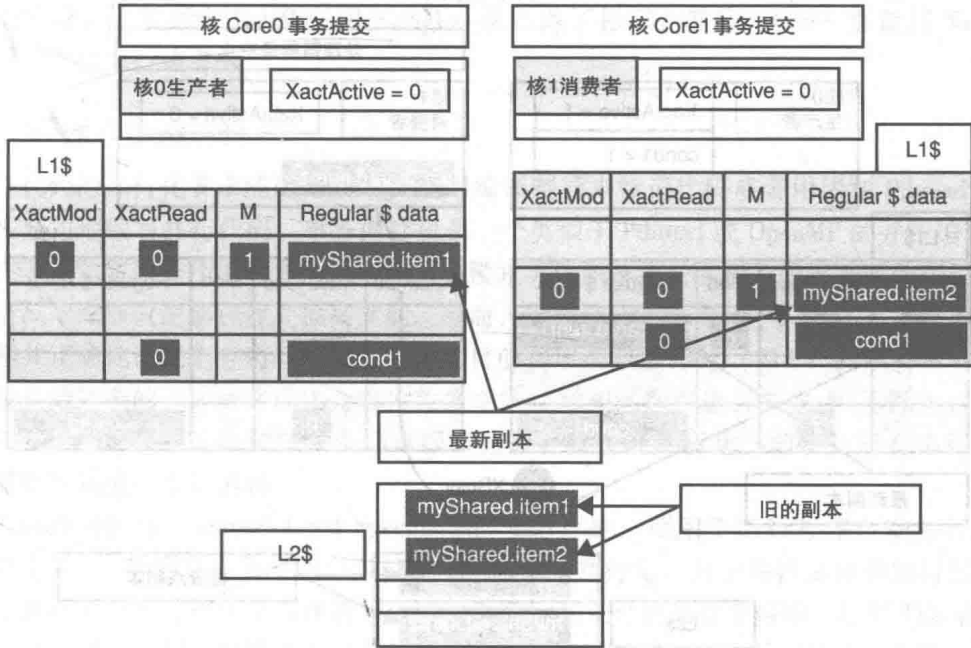
一旦全相联事务 cache 用完了所有的无效块或正常块, 事务将无法继续执行下去, 在这种情况下, 必须有一个能支持事务完成的供备选的原子执行机制。通常, 事务 cache 还可能在内存 (物理内存), 甚至是线程的虚拟存储空间内发生溢出。上述方法对短事务比较有效, 而对于长事务则软件开销可能很大。需要注意的是, 溢出问题也同样存在于设置了 transaction_read 位的 cache 块中 (内存和事务 cache 中的数据块都是这样)。不过, 在这种情况下, cache 只需要跟踪记录哪些块地址被读过了, 而不需要记录这些数据块的实际内容。

例 8.9 事务内存提交和中止示例 图 8-16 和图 8-17 说明了 TM 系统是如何工作的。在本例中, 再次使用图 8-15a 中的程序, 程序中使用了 TM 语法来描述生产者和消费者的功能。图 8-16 给出了 CMP 中的两个处理器核: Core0 和 Core1, 它们分别运行生产者和消费者的代码, 每个处理器核都有一个私有的 L1 cache, 每个 cache 行都增加了额外的位: XactMod (事务位) 和 XactRead (transaction_read 位)。

首先, 假定生产者中的 cond1 为真, 而消费者中的 cond2 为假, 因此生产者和消费者工作在不同的数据项上 (分别是 item1 和 item2)。在图 8-15b 中, 当两个处理器核都执行 transaction_begin 时, 处理器核中的 XactActive 位 (transaction_active 位) 被置位。然后, 两个处理器核分别读取变量 cond1 和 cond2, 假定对这两个变量的访问都在各自的本地 cache 中命中, 并且都处于 Shared 状态 (因此 L2 cache 中包含 cond1 和 cond2 的最新副本)。Core0 将包含 cond1 的 cache 行的 XactRead 置位, 同样, Core1 将包含 cond2 的 cache 行的 XactRead 置位, 通过设置 XactRead 位, 每个处理器核跟踪记录了当前活跃事务所读取的 cache 行。接下来, Core0 执行语句 “myShared.item1 ++”, 这个语句先读取 myShared.item1 变量, 然后再对其进行修改。首先,



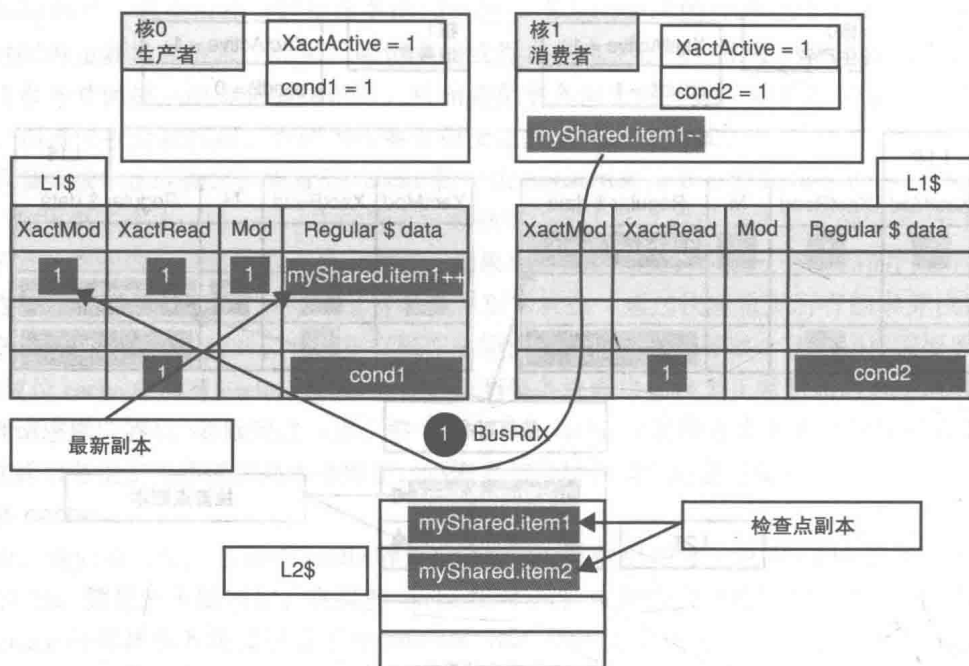
a) 执行中的事务



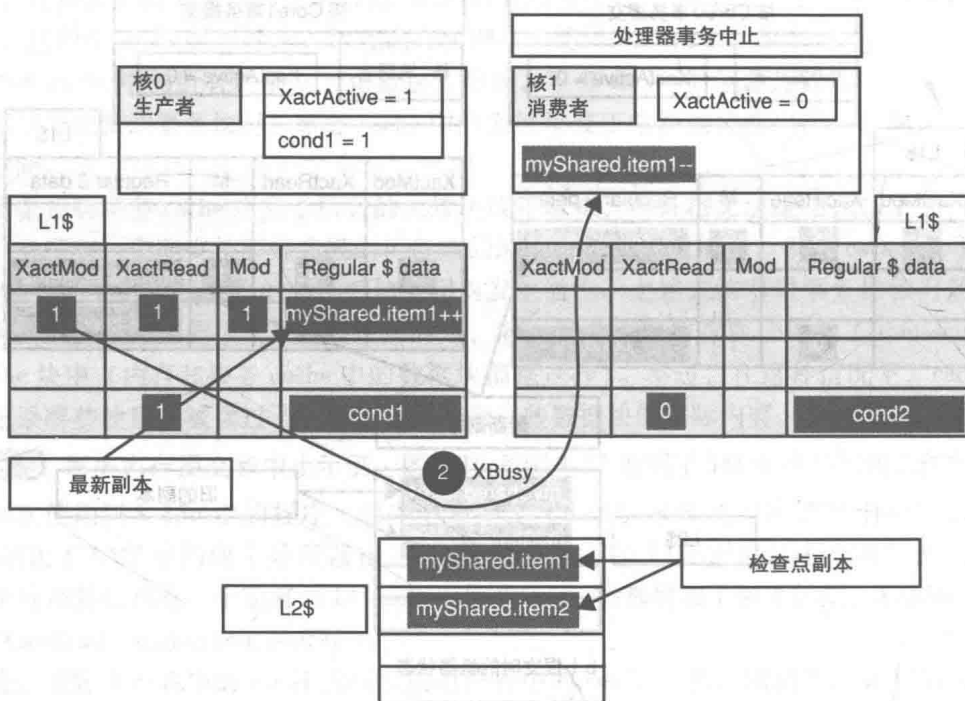
b) 提交时的机器状态

图 8-16 事务提交

Core0 为包含 item1 的数据块发送 BusRd 请求，一旦数据块副本加载到 cache 中，就将 XactRead 置位为 1。Core0 修改 myShared.item1 之前，先为其对应数据块发送一个 BusRdx 请求，Core1 按照一致性协议对其进行响应，如果 Core1 有共享副本，那么只需要简单将该副本无效掉即可，此后 Core0 就可以自由更新 item1 了。需要注意的是，只有当 Core0 的活跃事务提交时，该修改才对其他核可见，如果 Core0 的活跃事务中止，则必须取消该更新操作，除了置 M 位外，Core0 还需要置 XactMod 位，以表明此 cache 行已经在事务内部进行了修改。



a) 执行中的事务



b) 中止时的机器状态

图 8-17 事务中止

执行“myShared.item2 --”时，Core1 依次将对 item2 将 BusRd 和 BusRdX 请求发送到总线上，整个执行过程和上面 Core0 的执行过程相同。在这一过程中，Core1 将对其 cache 行的 XactRead 和 XactMod 位进行置位，此外，也将 cache 行的 M 位置位，这表明该数据块处于 Modified 状态，且在事务内已经进行过本地的读写操作，图 8-16a 给出了此时的机器状态。需

要注意的是,此时的 L2 已经存储了 item1 和 item2 的副本,而且都是事务开始前这些值的检查点版本。此时 L1 cache 中包含最新的副本,但仍是推测值,因为其对应的 XactRead 和 XactMod 都被置位了。目前情况下,当 cond1 = true 且 cond2 = false 时,两个事务并不冲突,因此 Core0 和 Core1 继续执行 transaction_commit,此时,两个 L1 cache 中的 XactRead 位都被重置为 0。此外,两个处理器核都将 XactMod 重置为 0,并将 M 位置为 1,以表明 item1 和 item2 的最新值在 L1 cache 中且 L2 cache 中的副本是旧值,之后,Core0 和 Core1 将其 XactActive 位进行重置,以表明退出事务模式,事务提交后的机器状态如图 8-16b 所示。

下面我们再考虑 cond1 和 cond2 都为真的情况,具体情况如图 8-17a 所示。假定 Core0 是第一个进入事务并执行 item1 递增操作的处理器核,正如我们之前解释过的,Core0 的 L1 cache 的状态如图 8-16a 所示。Core1 紧随 Core0 之后运行,最终也需要对 item1 进行递减。Core1 随后先发出一个 BusRd,就像之前介绍的那样,但主要区别在于这里只对 item1 发出 BusRd 请求,而不包括 item2。当 Core0 收到 BusRd 时,通过查看 XactMod 位的记录,Core0 知道另一个处理器核正试图读取一个被自己修改过的数据项,这也是 MSI 协议中正常操作和事务操作的不同之处。此后 Core0 只发送一个 XBusy 信号给 Core1,而没有发送最新版本的 item1,如图 8-17b 所示。一旦 Core1 收到 XBusy 信号,就意味着必须有一个事务中止。在这个例子中,收到 XBusy 信号的处理器核会中止其事务,因此,Core1 会重置所有的 XactRead 位,并通过设置 XactMod 位无效掉所有的 cache 行。请记住所有被无效掉的数据块所对应的检查点版本都在 L2 中,对这些数据块的最近更新都在 Core0 中用 XactMod 状态做了标记,然后,Core1 重置其 XactActive 位,稍后再重试执行该事务。

8.5.4 线程级推测执行

软件代码的并行化要求应用开发人员使用编译制导来标记代码或者用诸如 Pthread 这样的线程包来显式地编写并行代码。事务内存也是一个类似于 Pthread 或 OpenMP 的并行编程模型,其优点在于,非数值应用中的复杂线程间通信模式可以用事务来实现,而不需要使用锁,这在大部分情况下能够改进编程性并提高性能。然而,即使在 TM 的支持下,也还是需要程序员来识别线程并正确地设置事务边界。显式并行是目前提取线程级并行(TLP)的最有效方法,但事实上也是最复杂的。应用并行化会因为数据竞争(对相同数据进行并发读/写操作)而引入一些难以察觉的错误,这在调试器中通常很难重现。要实现并行化,即使已经有串行源代码了,也需要对其进行重新编码。

线程级推测执行(Thread-Level Speculation, TLS)是一项用于串行程序自动并行化的技术,借助于 TLS,程序员不需要做太多的工作就能完成并行化。由于并行是自动执行的,并行代码的正确性可以通过软件(编译器和运行时系统)和硬件的结合来保障。尽管 TLS 和 TM 都使用了类似的硬件机制,但是两者完全是针对不同目标的不同方法。在 TLS 中,程序员所需要做的所有事情就是识别出程序的特定区域,如循环、嵌套子程序或者递归函数调用,这些区域占据了串行程序执行时间的大部分,我们将其作为推测并行执行的候选。在找出这些区域后,剩余的工作由 TLS 软件和硬件来完成。

在常规串行代码中,TLP 的主要来源是嵌套的子程序、递归的函数调用以及循环。子程序和函数调用通常较难并行,因为子程序或函数调用通常需要返回值给调用者,如果调用者在为子程序创建一个线程后,立即推测执行,则它需要预测子程序的返回值,或当到达需要子程序输出值的位置时就停止执行。因此,利用嵌套子程序或递归函数调用来实现并行比利用循环迭代要困难很多,这也是传统的并行编译器专注于循环的原因。

循环并行化

一般情况下，循环迭代的并行化主要受到一些循环间数据依赖（loop-carried data dependency）的影响，而其他的相关处理则比较容易。由假相关或命名相关导致的数据冲突可以通过重命名来解决，而循环内的相关可以在每个线程内部解决。图 8-18 给出了 3 个循环片段，每个循环分别包含一些以不同方式影响其并行化的语句，在图 8-18a 中，for 循环中的语句没有循环间的相关，因此不会对并行有限制，这里唯一限制并行的是循环迭代的次数。在图 8-18b 中，对数组 A 的访问方式将可并行部分限制在 4 个循环迭代中，执行时每次只能以 4 次循环迭代作为一个“块”执行，其最大的并行加速比也就是 4。

<pre>main() { ... for(i:=m; i<M; i++) { ... A[i]:=A[i]+B[i]... } ... }</pre> <p>a)</p>	<pre>main() { ... for(i:=m; i<M; i++) { ... A[i]:=A[i-4]+B[i]... } ... }</pre> <p>b)</p>	<pre>main() { ... for(i:=m; i<M; i++) { ... j:=C[i] A[i]:=A[j]+B[i] } ... }</pre> <p>c)</p>
---	---	--

图 8-18 不同并行难度的循环示例

图 8-18c 的循环在编译时无法并行化，因为数组 A 的索引 j 在编译时是未知的，它包含在动态可修改的数组中。对某些迭代来说，索引 j 的值可能比索引 i 的值小，这在循环迭代并行执行时，会导致 RAW 冲突，在这种不确定的情况下，传统并行编译器会将循环转换为串行代码。然而，实际情况也可能是在所有迭代中，索引 j 都不会比索引 i 小，或者当索引 j 小于索引 i 时，两者之间的差距 i-j 也总是很大，以至于计算 A[i] 和 A[j] 的迭代永远不会在多核甚至是众核处理器中有任何时间上的重叠，因此编译器的这种简单处理方法可能会导致大量的线程级并行机会被浪费掉。

如图 8-18c 中的这类循环，在编译时无法证明它们是否真的没有循环间相关，这种情况在数值和非数值代码中是十分常见的。在数值代码中，如果循环中的数组索引是循环索引的线性函数，则只能证明在一段合理的时间范围内，不存在循环间相关。此外，在稀疏矩阵计算中，通过其他整型数组的值进行数组索引也是非常常见的。不过，在非数值应用中，通常会大量使用指针，因此最容易出现这种跨循环迭代的、无法静态确定地址的内存访问。最后，通常情况下，不确定性还有另一个来源，就是条件分支可能影响循环迭代内部的内存访问。

线程级推测执行可以支持所有循环的并行化，包括含有模糊的循环间数据相关的情况（参见图 8-18c 的例子）。对于循环的每个连续迭代，TLS 都推测创建出一个新的线程，并和原有的线程（在执行过去和现在的迭代）并行执行。线程是推测执行的，因为即使还有一些输入值没有确定，线程也会开始执行。因此，循环迭代之间可能会发生写后读相关引起的冲突。我们必须对这种冲突进行动态检测，当检测到冲突时，依赖冲突数据的线程（以及所有执行后续迭代的其他推测线程）的执行过程必须取消、回滚，并重新运行。当某个更早的迭代修改了某个内存地址，而该地址又被更晚的某个迭代读取时，就将出现写后读相关冲突。

并行循环代码中的内存冲突

图 8-19 展示了当试图并行执行多个循环迭代时所引发的内存冲突，图 8-19a 给出了循环中的前 4 个顺序迭代，图 8-19b 给出了 4 个线程的并行执行。线程的起始时间是错开的，因为它

们是按照循环索引顺序依次创建出来的。

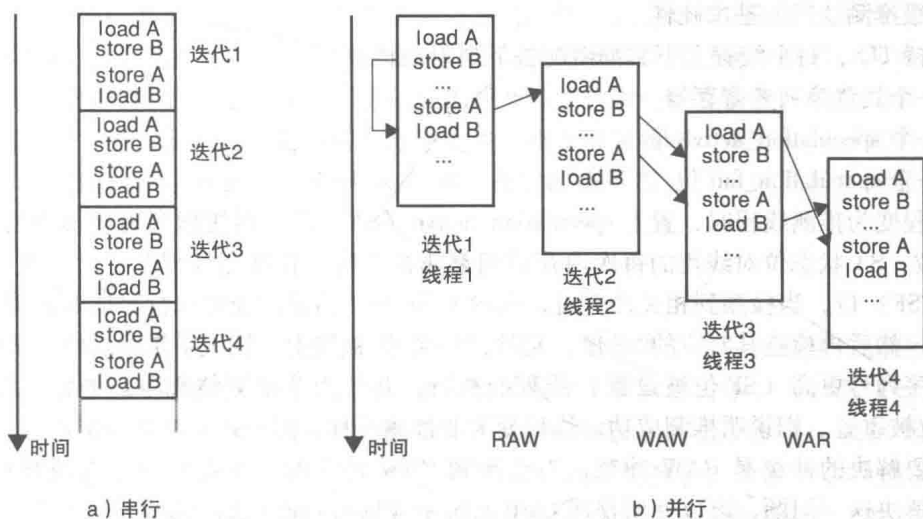


图 8-19 循环并行中的内存冲突

为了避免图 8-19 中显示得过于混乱，我们将不同类型的数据冲突（RAW，WAW，WAR），放在连续迭代的不同组中进行说明。如迭代 1 所示，每个线程中 load B 操作必须总是返回同一线程 store B 的值，与之相反，load A 操作必须总是返回之前线程 store A 的值。图 8-19 显示在迭代 1 和迭代 2 之间违反了这种相关性，因为在线程 1 执行完 store A 之前，线程 2 的 load A 就已经执行。这是一个 RAW 冲突，要解决这个问题，必须推迟线程 2 的执行使得 load 等待 store 完成，或者在检测到冲突后，取消线程 2 当前的执行，并在稍后重新执行。迭代 2 和迭代 3 中则展示了 WAW 冲突。按照语义，变量的当前值必须是按照循环串行顺序时的最新值，在图 8-19 的情况下，这一点是满足的，因为对每个变量的两次 store 都是按照循环索引顺序进行的，不过，这里仍然存在一个潜在的冲突，因为如果线程 2 由于某些原因被推迟执行 store，那么将导致线程 2 和线程 3 对 A 和 B 的更新操作乱序执行，从而违反 WAW 冲突。最后，迭代 3 和迭代 4 中展示了 WAR 冲突，对 A 的 WAR 冲突是满足的，但对 B 的 WAR 冲突就不满足了，因此，线程 3 中的 load B 操作将不会返回同一迭代中的 B 的值（这才是按照串行顺序约束下的最新值），而是会返回在迭代 4 中的 store 值，同样，我们也必须解决这样的冲突。

循环的推测并行

图 8-19 给出了并行循环中内存冲突的简单例子，但实际上，这样带有紧密循环间依赖的真实循环是无法有效并行的。因此，TLS 主要面向循环间依赖很少、间隔很远，且在编译时难以预测的场景，如图 8-18c 中的循环。

为了保证正确性，循环指令的执行效果看起来必须和按照循环索引顺序执行时一样。每个线程都分配一个序列号，线程被记为 T_i ， i 是按循环索引顺序的索引号。最旧的（头）线程的序列号最低，所有其他线程都是推测执行的，因为这些线程可能还和更老的且还在运行的线程之间存在 RAW 冲突，当动态检测到 RAW 冲突时，推测线程就必须取消。在图 8-19 中，线程 1 是头线程，线程 2、3、4 是推测线程，头线程必须在下一个线程变为非推测性线程之前，先将其所有执行结果提交到内存，一旦某个线程完成，其对应的线程上下文就可以给另一个新的推测线程使用。我们再看一下图 8-19 中的例子，当线程 1 的所有 store 都提交，且线程 1 退出后，线程 2 将变为头线程，此时，执行循环迭代 5 的线程 5 开始在之前被线程 1 使用并释放的硬件上下文上推测运行。因为头线程最终总是会变为非推测线程，所以这个计算过程总是可以往前

推进。

线程级推测执行的基本硬件

要支持 TLS，每个线程上下文都需配备下列附加状态：

- 一个线程序列号寄存器
- 一个 speculation_active 位
- 一个 speculation_fail 位

当线程变为推测线程时，置上 speculation_active (SA) 位，当线程变为非推测线程时，则重置 SA 位，SA 状态位对线程的每次内存访问都非常关键。在推测线程开始时，重置 speculation_fail (SF) 位，当检测到相关冲突时，则置上 SF 位。当推测线程执行完成时，需要等到变为头线程，然后再检查其对应的 SF 位。此时，如果 SF 被置上，则说明推测失败，头线程取消其所有序列号更高 (SF 位被设置) 线程的执行，并作为非推测线程重新开始执行。相反，如果 SF 位被重置，则说明推测成功，线程变为非推测线程，然后提交结果并退出。

关键要解决的冲突是 RAW 冲突，当检测到 RAW 冲突时，主要问题是在线程中设置 SF 位。为了解决这一问题，可以使用普通 CMP 系统中常见的 cache 层次结构，这与 TM 中的情况类似，L1 cache 保存推测值，而共享 L2 cache 只保存已经提交的 (非推测的) 值。除了 RAW 外，WAW 和 WAR 冲突可以通过在 L1 cache 中进行重命名来自动解决 (即每当推测线程访问一个内存块时，就在本地 L1 cache 中分配一个推测的数据块副本)。

常见的 MSI-无效协议通过每个 cache 行的两个状态位 (有效位和脏位) 以及两种总线请求 ((BusRd 和 BusRdX) 来确保 L1 cache 之间的一致性。在支持 TLS 时必须将 MSI-无效协议中加入一些新的特征，以便处理 L1 cache 中的推测值的问题。对推测 cache 块的修改必须限定在本地 L1 cache 中，并且不能传播到存储系统中 (比如 L2 cache 或其他 L1 cache)。例如，决不允许将数据块的推测副本进行刷新，只有非推测修改的副本才能转发给其他 L1 cache。

为了检测 RAW 冲突，每当对推测数据块进行修改时，都必须通知远程 L1 cache，因此，在协议中增加了一个名为 Squash 的总线请求。每当 store 在推测数据块中命中时，就会发送一个 Squash 总线请求到所有 L1 cache，线程的序列号会被打包到 Squash 总线请求上，这样每个 cache 都可以判断自己相比于发出请求的 cache，是处于更靠前的推测还是靠后的推测。当推测线程从更低序列号的线程中接收到一个 Squash 请求时，线程的 SF 将被置位，该线程中止，后续的所有其他线程也都被取消。

推测数据块的副本必须一直留在 L1 cache 中，直到线程的推测执行阶段结束，因为这样可以表明该数据块被推测加载和修改过，并且有对数据块副本的推测修改的跟踪记录。如果执行过程中需要将该数据块副本替换出 cache，那么它对应的线程以及后续的所有线程都必须取消，就和检测到相关冲突的效果一样。最后，线程一旦变为头线程和非推测状态，那么线程就可以成功执行了。

当推测线程已经执行到最后，并变为头线程后，就将检查 SF 标志位。如果 SF 还是处于重置状态，则表示推测成功，此时，L1 cache 中的所有推测数据块必须先变成非推测的，然后线程退出，下一个线程再变为头线程。如果 SF 位被置位，则推测失败，此时所有的推测数据块均被无效掉，然后线程在非推测模式下重新执行代码。

优化

上述的硬件支持描述非常简单，它面向的是真正具有循环间依赖的情况，并且依赖关系通常具有较大距离、且不可预测。当存在相关的两次访存之间的地址很少相等 (或从不相等) 时，这种方式就非常有效。TSL 还可以和循环迭代间的同步操作结合起来使用，以消除冲突。当相关的地址是相同的，或大部分情况下是相同的，并且相关距离很短时，编译器会尽量去避

免冲突或者是使用同步来消除冲突。这里举一个冲突避免的简单例子，在每次迭代中都有这么一个操作：更新循环索引，然后检测索引值看是否循环已经执行结束。对于这种情况，我们可以简单地将循环索引更新并从循环主体中移除，让其成为推测线程创建机制的一部分。

还有很多其他可行的优化，但通常每种优化都会增加软硬件的复杂度。推测线程一旦发生冲突，就可能会立即重启，而不会等到执行完成并变成非推测状态。还有一些更先进和复杂的优化策略，可以做到只取消那些受 RAW 相关冲突影响的执行部分，并只重新执行那些受影响的部分。

8.5.5 帮助线程

TM 和 TLS 都依赖有效的硬件支持来检测线程间的访存冲突，并中止预测错误的线程，此外，这些推测线程必须在确认推测成功后才能将数据提交到架构状态中，这导致这些方法在具体实现时更加复杂。如果推测线程不会对任何架构状态做修改，那么复杂性可以有所降低。这时非推测线程负责执行程序，而推测线程仅仅用于辅助程序的执行，推测线程在性能、功耗、可靠性等方面可能会有所帮助，这种线程通常叫作帮助线程、辅助线程或从属线程。

我们使用数据预取来说明帮助线程是如何工作的，因为预取是帮助线程最常见的应用。数据预取在应用需要数据之前就推测性地将数据从内存预取到 cache，这是一个无约束力的 (non-binding) 访存 load 操作，不改变结构状态，它只是将内存块取到处于一致性状态下的某个处理器核的 cache 中。为了更有效地预取数据，帮助线程在应用实际取出数据前就需要知道数据访问的地址，从而有足够的时间来及时完成预取。最后，预取地址必须准确，即预取的数据必须被应用程序使用，否则预取就白做了。大多数预取机制是基于硬件实现的，简单且自动完成，一般在出现 miss (失效) 之后的地址空间中连续预取多个数据块 (顺序预取)，或者通过固定的步长来预取数据块 (步长预取)。不过，在如今的主流应用领域，包括数据库、多媒体和游戏等应用中，内存访问范围都很大，通常很难有效利用处理器的 cache 结构，往往存在大量的 cache 失效，导致明显的性能下降。这些应用的数据访问行为不可预测，因此要准确预测不久的将来会被访问的地址是非常困难的，甚至是不可能的。因此，帮助线程不是基于某种预测器来进行地址预测，而是预先计算出这些难以预测的地址再进行预取。通常帮助线程中实现的预取算法可以是每个数据访问专门定制的，因为这种机制不是基于硬件的。

为了预先计算 load 访问的地址，与该 load 操作相关的帮助线程先执行能够计算出该 load 地址的一个程序指令子集，称为 load 操作的反向切片 (backward slice)。图 8-20 说明了什么是反向切片，从指令 I6 开始，程序执行反向跟踪，寻找之前的最近一个定义 R2 的指令，在这个例子中，指令 I5 通过寄存器 R3 和 R5 的加操作来定义 R2，继续反向跟踪程序的执行再找到对 R3 和 R5 的最近定义，可以看到 R3 由指令 I3 定义，R5 由指令 I1 定义，然后上述过程继续递归下去。在本例中，下一步操作就是继续寻找定义 R4 值的指令。



图 8-20 帮助线程示例

一旦 load 操作的反向切片形成, 其对应的指令将包含在帮助线程中, 由于反向切片包含的指令比整个程序少, 帮助线程执行反向切片的速度很可能比主线程的速度快, 因此帮助线程可能在获得 load 操作地址后, 在主线程执行 load 操作前的某个合适时间点将值返回到 cache 中。

上述方法中, 必须为每个需要预取的 load 操作产生反向切片, 由于反向切片的产生需要通过程序的依赖图来向后倒推, 因此只能在编译时完成。此外, 还应该尽量减少需要产生预取帮助线程的 load 操作的数量。在大多数应用中, 出现 cache 失效的 load 操作占全部 load 操作的比例一般都比较小。通过分析, 编译器可以识别出那些可能会出现失效的 load 操作, 这类 load 通常被称为失职 load (delinquent load), 然后编译器为这些 load 产生反向切片。反向切片中的指令可能会有限制, 这使得编译器在产生反向切片时, 在依赖图中不能回溯太远。一旦编译器停止产生反向切片, 就需要将切片中使用了但是不在切片中定义的寄存器值作为切片的输入, 在本例中, R4 的值需要作为切片的输入。

一旦切片产生, 就可以存储为应用程序二进制文件的一部分, 编译器甚至可以通过一条专用的制导指令来指示何时启动帮助线程。例如在这个例子中, 编译器可以在指令 I1 处放置一条制导指令来启动帮助线程, 使用针对运行时环境的制导指令给出切片的输入寄存器。帮助线程的硬件根据编译制导指令来决定是否启动帮助线程。例如, 如果当前 CMP 中没有空闲的线程上下文来执行帮助线程, 硬件可能会忽略上述产生帮助线程的编译制导指令; 此外, 即使有可用的线程上下文, 如果动态条件不是很理想 (比如由于总线拥塞, 目前系统中已经有很多未完成的访存操作), 硬件也可能会决定不启动帮助线程。如果帮助线程与主线程在同一个多线程处理器核上执行, 则无需额外将数据取至 L1 cache 中。如果帮助线程和主线程在 CMP 的不同处理器核上执行, 则切片的输入寄存器值必须前递给帮助线程。此外, 大多数 CMP 处理器核只共享 L2 cache, 因此, 当帮助线程与主线程运行在不同的处理器核上时, 也只能将内存数据预取到共享的 L2 cache 中。

为了更好地发挥预取帮助线程的作用, 还有几个选择是必须谨慎考虑的。首先, 帮助线程是作为一个专门函数调用的, 在距离目标 load 足够远之前就启动执行, 以此来隐藏潜在的 cache 失效延迟。然而, 如果这个距离太长, 那么预取回来的数据可能在 load 执行前又被替换掉。其次, 反向切片结构在存在模糊数据依赖的情况下可能会不准确, 从而导致不准确地计算 load 地址。最后, 帮助线程也会产生额外的功耗来执行切片程序, 因此, 帮助线程必须能够有效减少 cache 失效次数, 以此来补偿帮助线程自身的开销。

8.5.6 通过冗余执行提高可靠性

传统情况下, 系统可靠性主要通过双模或三模冗余来保障。在三模冗余情况下, 同一份代码在三个处理器上锁步执行, 每个周期后, 对三个处理器执行的结果进行比较, 并通过多数优先的投票方式来决定三个结果中的哪个应该被提交到架构状态中。很显然, 对于日常的低端计算来说, 这种重复执行的开销肯定是难以接受的。不过, 由于晶体管尺寸的缩小和电源电压的降低, 可靠性在低端计算中也越来越重要, 我们可以借助 CMP 和多线程处理器核中的大量线程来进行冗余执行, 从而提高系统的可靠性。

片上多处理器在多个粒度上提供了冗余组件, 例如在同构 CMP 中, 两个处理器核完全相同, 除了共享 L2 cache 之外, 没有任何其他计算结构的共享。在多线程处理器核中, 线程上下文共享多种资源, 如功能单元和译码器, 不过寄存器文件和重排序缓冲区可能是分离的。组件上的冗余设置可以用于提高系统的可靠性, CMP 中的两个处理器核或多线程处理器核中的两个线程上下文可以同时运行相同的代码, 其中一个线程是领导者以及执行程序的主线程; 另一个线程是检查者, 对两个处理器核或同一核上的两个线程上下文的计算进行相互校验。当检查

者发现两次执行存在不一致时，则检测到错误，该错误可以通过后面重新执行来纠正。由于架构状态还没有更新，这两个线程可以在引发错误的指令处重新执行。比如，这两线程可能清空流水线，并从引发错误的指令处重新取指。一方面，如果第一次检测到的错误属于瞬时错误，那么再次执行时基本不大可能再发生这种错误。另一方面，如果错误是永久性的硬件错误，则重新执行并不能解决问题，但至少能够检测到该错误，并采取必要的纠正措施。

当利用核内多线程来执行两份相同代码时，整体可靠性受到两个线程间资源共享程度的限制，如果共享结构中存在永久性或瞬间故障，会同时影响两个线程的执行，但这种情况下的错误无法检测到。例如，如果两个线程上下文共享译码逻辑，那么译码器中的永久故障将导致指令执行的两个实例执行结果完全相同，但却是错误的结果。

图 8-21 给出了核内多线程是如何检测硬件故障所造成的错误的，在此例中，两条指令在双路交错多线程处理器核上的两个线程中冗余执行，其中一条指令是一个简单的整数加法指令 ($R1 = R2 + R3$)，另一条指令是一个复杂的浮点除法指令 ($F1 = F2/F3$)。指令预取队列 (IFQ) 包含这两条指令，每条指令有两个实例，多线程处理器核有两个译码器：一个是简单译码器，只可以译码整数指令；另一个是复杂译码器，既可以译码整数指令，也可以译码浮点指令。整型 ALU 也有两个，但是浮点单元 (FPU) 只有一个。当第一个整数指令冗余执行时，两个译码器和 ALU 并行使用，由于对这条整型指令的两次执行不共享逻辑块，因此冗余执行可以检测到任意一个执行中出现的错误；又由于执行中用到的两套硬件逻辑也完全相同，这种情况下也可以检测到硬件故障。

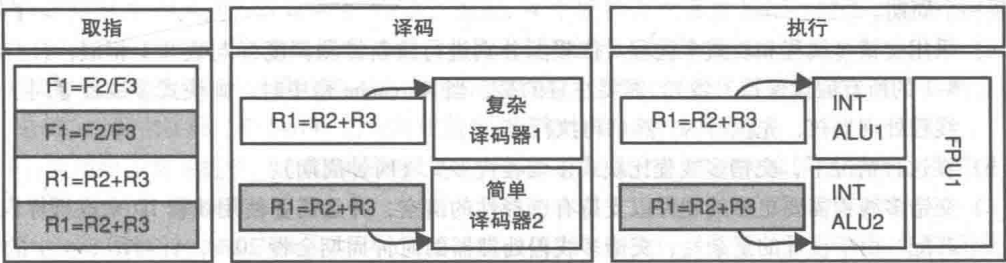


图 8-21 基于同时多线程处理器核空间冗余执行的硬件故障检测

下面我们再看第二条浮点指令的执行情况，如图 8-22 中所示。此时，只有复杂译码器能够译码这两条浮点除法指令（一个原始副本，一个冗余副本），原始副本和冗余副本以时间冗余的方式执行，即浮点指令的第一个实例在第 N 个周期译码，而第二个实例在第 $N + 1$ 个周期译码。因此，瞬时错误几乎一定能检测到，因为这两次执行基本上不可能都碰到同一个瞬时错误。不过，复杂译码器中的硬件故障会以同样的方式影响两个副本执行，因此，硬件故障无法检测出来。同样，由于两个副本都在相同的 FPU 上执行，因此也无法检测到 FPU 单元中的硬件故障。

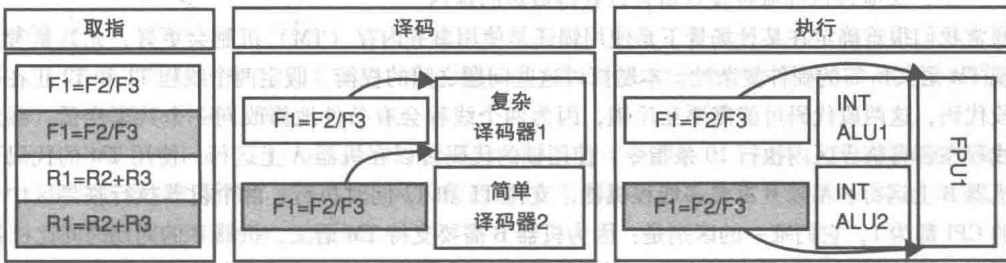


图 8-22 同时多线程处理器核因共享资源而无法利用冗余执行检测到硬件故障

习题

- 8.1 本题探讨是否值得增加新的改进微架构以提高块式（粗粒度）多线程处理器的性能（不考虑功耗）。假定线程切换只发生在 L2 cache 失效时，且开销为 60 个周期（新线程可以开始执行之前的时间）。再假定 cache 失效时，有足够的就绪线程用来切换，且已知当前的 L2 cache 的命中率是 50%。新的微架构模块是 cache 命中预测器，可以预测访存操作是否在 L2（注意不是 L1）cache 中命中。我们用这个预测器来决定何时进行线程切换。如果预测器预测 cache 失效，则尽早启动线程切换。考虑如下 4 种情况：
- (a) 预测器预测 L2 cache 失效，真实结果也是 L2 cache 失效，此时线程切换会尽早启动，且线程切换开销可以降低至 20 个周期（对比基线 60 个周期）。
 - (b) 预测器预测 L2 cache 失效，真实结果是 L2 cache 命中，此时由于之前启动了线程切换，会导致不必要的流水线清空，最终导致线程切换开销增加到 120 个周期。
 - (c) 预测器预测 L2 cache 命中，真实结果也是 L2 cache 命中，此时没有启动线程切换，因而没有收益也没有损失。
 - (d) 预测器预测 L2 cache 命中，真实结果是 L2 cache 失效，此时失去了尽早启动线程切换的机会，因此将付出 60 个周期的线程切换代价。
- 针对上述 4 种情况，请问在什么样的预测精度下，这种新的微架构模块可以达到一个损益平衡点，如果 L2 cache 的命中率从 50% 提高到 80%，那么对预测器精度的影响又是如何？
- 8.2 使用图 8-4 中所示的依赖图来解答本题，我们再做出如下修改：假定 X5 cache 命中，因此计算只需要一个周期。
- (a) 采用交错多线程和块式多线程对依赖图分别进行重新推测调度（与表 8-1 相似，且与生成表 8-1 的所有假设保持一致），需要注意的是，当 X5 cache 命中时，则块式多线程等同于一个单线程处理器核，先执行 X，然后再执行 Y。
 - (b) 在这种情况下，交错多线程比块式多线程快多少（时钟周期）？
 - (c) 交错多线程需要更多的硬件以支持有选择性的清空，并且需要使用线程 ID 来进行标识（tag）匹配。由于设计的复杂性，交错多线程处理器的时钟周期会慢 20%。针对图 8-4 中的线程代码（X5 cache 命中），使用交错多线程来执行是否还有意义？
- 8.3 假定有三种片上互连网络的系统架构可以选择：单向环，双向环， $N \times N$ 的 mesh 网络。L1 cache 之间的一致性有两种选择：基于侦听的，基于目录的。现有一个包含 64 个处理器核的芯片，每个处理器核带一个私有的 L1 cache，且所有核共享一个较大的 L2 cache，L2 cache 分为 64 个 bank，每个 bank 分配到一个处理器核上，两个直接相连的处理器核之间的通信延迟为 1 个周期，路由延迟的计算方式是数据包每经过一个路由器就增加 1 个周期，基于目录的 cache 一致性机制下访问目录的时间是 16 个周期。假设不存在任何线路或目录访问的竞争。
- (a) 选择哪种设计组合时，维护一致性所需的延迟最短？
 - (b) 假设有一个片上 1024 核的众核 CMP 结构，目录访问的延迟也增加到 20 个周期，在这种情况下，又应该选择哪种设计组合以获得最短的延迟？
- 8.4 通常我们很难确定在某种场景下是使用锁还是使用事务内存（TM）机制会更好，尤其是考虑到实现 TM 语义所需的硬件复杂性。本题探讨这些问题之间的权衡。假定两个线程 T1 和 T2 正在执行两段代码，这两段代码可能需要互斥锁，因为两个线程会有条件地修改同一个共享变量。假定每个线程在各自临界区内执行 10 条指令。使用锁的代码可以在机器 A 上运行，使用 TM 的代码可以在机器 B 上运行。A 和 B 都是多线程机器，支持 T1 和 T2 同时执行，两个机器执行临界区中代码时的 CPI 都为 1，它们唯一的区别是：因为机器 B 需要支持 TM 语义，机器 B 的周期时间比机器 A 的周期时间长 20%。如果事务失败，则额外需要 20 个周期开销来清除事务状态。
- (a) 如果两个线程同时读取/修改共享变量的可能性是 10%（即 90% 的时间没有共享），哪个机器更快？

(b) 随着临界区的大小逐渐增大, 出现冲突的可能性也逐渐增加, 因此, 如果临界区中指令数从 10 增加到 1000, 出现冲突的可能性从 10% 增加到 20%, 在这种情况下, 又是哪个机器更快?

- 8.5 假定一个 16 路 CMP 系统运行在一个功耗受限的环境中, CMP 系统运行时可以自动地配置为 1, 2, 4, 8, 16 个处理器核, 但总的功耗限制是固定的。例如, CMP 系统可以按单核形式运行, 让其他 15 个处理器核都休眠, 从而借助其他休眠线程所节省的功耗来提高自身的频率。假定休眠和唤醒的时间开销都为零, 而功耗和频率呈平方关系。比如, 当单个处理器核可以使用全部 16 个核的功耗时, 其频率就可以提高 4 倍。上述这种结构称为 EPI 可调 CMP。

现在假定有一个可部分并行的应用, 启动时是单线程, 开始的串行模式占据 5% 的执行时间; 在接下来 40% 的时间里, 该应用以 16 个线程运行; 而在接下来的 40% 的时间里, 以 4 个线程运行; 在剩余的 15% 执行时间里, 又是单线程。

- (a) 相比单核机器 (消耗的总功耗相同, 但是运行频率更高, 按照功耗与频率呈平方关系来计算), 这个 CMP 系统使用相同功耗运行该应用的加速比是多少?
- (b) 相比传统的 16 路 CMP 系统 (不支持处理器核的动态重配置功能), 题中的 CMP 系统使用相同功耗时能获得的加速比是多少?
- (c) 由于电压调节的限制, 假设将来的功耗与频率之间呈线性关系, 请解释在这种情况下, EPI-可调 CMP 还有哪些优势。
- 8.6 考虑图 7-2 流程图所示的四迭代 Jacobi 方法, 在这个流程图的每次迭代过程中, 4 个线程同时计算向量 X_i 的新值 (和 Y_i 的值是线性函数关系), 然后, 线程使用 barrier 进行同步, 接着使用 barrier 前产生的 X_i 值来并行计算 4 个 Y_i 值, 然后, 4 个线程再次通过 barrier 同步, 等待进行收敛测试。如果满足某个收敛条件, 则程序退出, 否则循环继续, 重新回到计算 X_i 。
- (a) 首先, 请使用 barrier 同步原语将流程图转换成并行算法的伪代码 (类似于图 5-2 所示的伪代码), 假定在第二个 barrier 后面的收敛测试函数在单线程中执行。
- (b) 使用编译制导指令, 将流程图转换为 OpenMP 并行程序, 可用的编译制导指令及其语义可以参考 www.OpenMP.org。
- (c) 将流程图转换为基于事务内存的实现, 代码作为事务执行, 类似于图 8-15b 中所示的例子。
- (d) 哪种实现方式 (OpenMP 还是事务内存) 可能会更快? 请分析这两种实现方式的性能。
- 8.7 考虑如下代码片段:

```

R3 = R7 × R4
R4 = R3 × R4
R7 = R5 × R6
R5 = R4 × 2
R3 = R5 × R4
Load R3, 0(R5) -- (L1)
Store R2, 0(R2)
R7 = R6 × 2
R2 = R3 + R5
Load R1, 0(R2) -- (L2)

```

代码最后的 load 指令 (L2) 是一个关键 load, 在 cache 中的命中率为 50%, 我们的目标是使用反向切片对该 load 进行预取。

- (a) 假定第一次 load (L1) 在 cache 中总是命中, 且命中延迟为 1 个周期。其他所有指令的执行也都只需要 1 个周期。如果 L2 的失效延迟是 10 个周期, 那么为了完全隐藏 L2 的失效延迟, 预取需要在多少条指令之前启动?
- (b) 假定第一次 load (L1) 在 cache 中失效的概率是 20%, 请说明在这种情况下, 切片中的预先计算是否有助于提高整体性能。

8.8 (a) 考虑一个简单的单线程五级流水线。该流水线将每一个 cache 失效视为一个冲突并冻结流水线, 当执行一个基准测试程序时, 假定每 100 个周期发生一次 L1 cache 失效, 如果该块是在 L2 上, 则每一个 L1 cache 失效需要 10 个周期来满足, 如果在 L2 上失效, 则需要 50 个周期。经过 200 个周期的计算后, 发生一次 L2 cache 失效, 假定在不考虑 cache 失效的情况下, CPI 为 1, 当考虑到 cache 失效延迟时, 实际的 CPI 是多少?

(b) 考虑与 (a) 相同的例子, 但假定硬件现在是两路多线程, 类似于图 8-3 所示。假定切换开销是零, 且现在有两个如第一种情况所述的具有相同 cache 失效行为的线程, 则此时在两路多线程机器上的两个程序的 CPI 分别是多少? CPI 是否有提高? 如果有, 解释是怎样提高的; 如果没有, 解释为什么要研制此两路多线程机器。

(c) 重新考虑 (a) 中的情况, 但线程切换开销是 5 个周期。再计算每个线程的 CPI, 并解释为什么它会增加、减少或保持不变。

(d) 考虑 L2 失效的延迟从 50 升至 500 个周期, 切换开销从 5 升至 50 个周期, 计算此时机器的 CPI。

8.9 以下两个改进的组合能够提高片上多核处理器的性能: (1) 增加更多的处理器核; (2) 增加更多的共享 L2 cache。基础芯片上有 3 个处理器核和 9 个 L2 cache bank, L2 cache 的大小可以通过添加 cache bank 来逐渐增大, 并且每个 cache bank 的面积都是处理器核面积的 3 倍, 我们已知如下内容:

(a) 工作负载的 60% 可充分并行, 其余的不能;

(b) 在四处理器核和四 cache bank 的基本配置下, 由于 L2 失效而造成的处理器核阻塞时间占每个处理器核执行时间的 30%;

(c) 为了保持相同的命中率, 每个处理器核共享 L2 cache 的数量应保持为常数;

(d) 仿真模拟显示, L2 的失效率随着每个处理器核大小的平方根而下降。我们推测即每个处理器核的阻塞时间也将随着每个处理器核大小的平方根下降。

假定你所在的公司已经获得了新的技术来构建大型芯片, 从而使下一代芯片的面积是目前芯片面积的 4 倍, 用以放置更多的处理器核和 L2 cache。根据你所知道的内容, 你会提出怎样的最好“初步”设计? 设计的特征具体由处理器核的数量和 L2 cache bank 的数量来描述, 它们可以是任何整数。设计应包含在新的芯片中, 试评估采用新的芯片面积的最好设计所带来的加速比。

8.10 本题是在共享 L2 cache 的 CMP 上, 研究维护 L1 cache 之间一致性的目录替换组织形式, 以下是已知的架构信息:

- CMP 拥有 8 个处理器核和 8 个 L2 cache bank, 它们之间通过交叉开关互连;
- L2 cache 的每个 bank 是 3MB 大小的 12 路组相联;
- 每个处理器核都有一级指令 cache。一级指令 cache 是直接映射的, 大小为 32KB;
- 每个处理器核都有一级数据 cache。一级数据 cache 是 4 路组相联的, 大小为 64KB;
- 所有 cache 中块的大小都是 64B;
- 内存的大小是 4GB。

基于上述数据, 试比较与共享 L2 cache 相关的维护 L1 数据 cache 间一致性的 3 个目录组织形式。假设指令是不可修改的, 可以缓存在共享 L2 cache 中, 但不会在数据 cache 中缓存。3 个目录分别是:

- 存在标志向量目录;
- 包含 L1 缓存目录 (标识位加状态位) 的 L1-map 目录;
- 图 8-12 所示的 L1-map 目录结构。

对它们进行比较, 分别计算每种情况下目录位的总数。

量化评估

9.1 概述

现代计算机系统集成了越来越多的器件和功能,这使得其复杂度急剧增加。对计算机体系结构研究者来说,在整个系统设计周期中,软件模拟是用来评估新创意和探索设计空间的极其重要的工具。相比于硬件原型系统和分析建模,软件模拟实现了精度、成本、灵活性和可靠性之间更好的平衡。由于当前微处理器设计复杂度的持续增加以及生产成本的急剧飙升,计算机体系结构模拟已变得至关重要。

软件模拟在计算机体系结构研究和设计过程中无处不在,它在很大程度上影响了研究与设计的生产率,主要体现在两个层面:(1)开发模拟器所花费的时间和精力;(2)在模拟器上运行代表性基准测试程序的时间开销。微处理器片上集成密度的快速增长为架构师提供了充足的片上资源,借此可以设计更复杂的系统结构以提高其计算能力。此外,功耗和可靠性也已经成为关键的设计约束,在模拟器开发中,要想构建一个能够帮助设计人员在一个统一框架下权衡性能、功耗以及可靠性的模拟基础架构平台,是需要大量成本和时间的。设计这样一个复杂的模拟基础架构的另一个直接后果是导致模拟速度变慢,进而会增加每次设计探索所需的时间。随着片上多处理器时代的到来,模拟速度变慢的问题变得更加严重。当前,用单线程来模拟核数不断增长的 CMP 系统方法是难以扩展,也难以持续。如果还像过去模拟单核系统一样继续用串行方式模拟 CMP 系统,那么模拟所需的时间必然会急剧增长。认识到这个问题的重要性后,目前已经有一些加快模拟速度以及更有效的系统建模方法,本章的主要目标就是介绍各种模拟方法,并探索加速模拟的主要途径。

本章主要介绍针对计算机系统设计架构解决方案进行有效量化评估的方法。计算机系统设计时需要考虑一些实际约束,例如芯片面积、功耗、散热以及可靠性等,大多数的创新解决方法都是在不同的约束之间进行权衡。因此,系统架构师的目标是了解如何在满足特定约束条件(如成本)的同时,最好地实现所期望的性能、功耗、散热以及可靠性目标。模拟也需要考虑类似的折中,目前已经有很多的模拟方法,但每一种方法都需要考虑模拟速度和模拟精度的折中。

本章我们首先介绍模拟的一些分类,模拟器可以从不同的维度进行分类:用户级模拟器与全系统模拟器;功能模拟器与时钟精确模拟器;trace 驱动模拟器与执行驱动模拟器;等等。注意上面这些分类都是正交的,我们可以从上面的每个分类维度中选择一些组件进行集成,从而构成一个完整的模拟器。目前有两种主要的模拟器集成方法:时序优先和功能优先。本章对这两种方法进行了详细的论述,以突出它们的相对优点。模拟器可能是单线程的,也可能是多线程的,在单线程模拟器中,只存在一个宿主机线程用来运行所有模拟任务。考虑到 CMP 系统已经非常普及,接下来也将介绍并行模拟,在并行模拟中模拟器本身是多线程的,每个模拟器线程模拟目标处理器核的一个子集,或者是模拟行为的一个子集。我们将介绍一些采用不同同步机制的并行模拟方法,这些同步机制需要在精度和速度上进行折中,通常是牺牲一定的模拟精度来换取更快的模拟速度。

由于对系统体系结构的模拟速度很慢,要想在当前宿主机上完整运行最新的基准测试程序

已经变得越来越困难,因此,出现了基于采样的技术,它能够选取出最能代表整个基准测试程序执行特征的一个小的执行片段。

模拟的出发点都是对我们感兴趣的某个指标进行量化,比如目标机上运行负载时的CPI的值。但仅仅获得某个指标的量化值是不够的,这只是系统设计的第一步,接下来需要理解在不同负载特征刻画下所观测到的行为背后的原因,对负载特征的刻画使得设计人员可以理解感兴趣的负载与负载所运行的目标系统之间的相互作用。本章最后将对负载特征刻画的方法进行简要的介绍。

这一章涉及大量的工具,这些工具的详细介绍可以在网上进行查阅,本书网站上给出了参考列表。

本章的主要内容如下:

- 9.2节介绍模拟器的分类方法——用户级模拟和全系统模拟;功能模拟和时钟精确模拟;trace驱动模拟、执行驱动模拟和直接执行模拟。
- 9.3节介绍如何在一个模拟器中集成多种模拟方法。
- 9.4节介绍模拟多处理器目标系统的两种方法——单线程多处理器模拟和多线程多处理器模拟。
- 9.5节介绍功耗和热量模拟。
- 9.6节介绍采样加速模拟的方法,包括SimPoint和系统随机采样方法。
- 9.7节介绍用于理解系统架构与负载之间相互作用的负载特征刻画。

9.2 模拟器分类

计算机体系结构模拟通常包含两个方面:模拟器和基准测试程序。模拟器模拟目标机的主要特征或行为,根据不同的设计目标和约束,在具有不同粒度和精度的级别上对目标处理器进行模拟。基准测试程序是一组在目标处理器上运行的程序,用来评估目标机的一个或多个组件。借助于准确定义的基准测试程序,可以公正地对各种体系结构或结构特征进行比较。设计和使用计算机体系结构模拟器需要在较高的模拟精度、较快的模拟速度以及较低的开发难度等不同目标上进行权衡,而这些目标往往是相互冲突的,尽管模拟工具开发人员对这些目标都很关注,但没有哪个模拟器可以同时满足这些目标,计算机架构师必须对目标进行优先级区分,以便更好地选择相应的模拟工具。而模拟器可以根据它们的模拟内容和模拟粒度进行分类。

9.2.1 用户级模拟器和全系统模拟器

处理器在现代计算机系统中扮演着重要角色,并且其结构复杂,因此处理器成为计算机体系结构研究和设计的首要目标。用户级模拟器注重模拟处理器的系统微架构,而不考虑一些系统组件,如协处理器和I/O设备。基准测试程序只模拟用户代码,并在模拟器上运行。当基准测试程序需要访问并未进行模拟的系统资源(如I/O设备)时,我们只对会影响资源分配的那部分架构进行简单功能模拟,而忽略微架构上的影响。例如,如果系统调用更新架构寄存器,该寄存器更新是在模拟过程中进行的,但是系统调用的过程被视为黑盒,模拟器中无法测量这一过程中对系统微架构的影响。

虽然用户级模拟器忽略了一些重要的系统影响,但仍然能够运行实际的应用程序,它拥有接近完整的工作环境,并且尽可能忠实地维持负载的真实性。在假设某些忽略不会损害模拟结果可信度的基础上,模拟器可以只关注处理器设计,而选择性地忽略一些子系统的模拟方法来简化建模的工作量。在大多数情况下,操作系统的活动是被忽略的,因此,用户级模拟器有时也会产生一些无法接受的误差。然而这种设计相对简单,减少了开发工作量,并且容易使用,这些特点在计算机体系结构领域都是非常具有吸引力的。

全系统模拟器模拟整个计算机系统，包括 CPU、I/O、磁盘以及网络等。它们能够启动和运行不加修改的现有操作系统，因此在模拟过程中比较容易捕获工作负载和整个系统之间的交互。由于应用的驱动，高端计算已经从传统的科学与工程应用扩展到新兴的信息处理应用，例如数据库、决策支持以及 Web 搜索引擎等，与此同时，全系统模拟器也变得越来越重要。上述商业应用存在更多的系统活动，在模拟过程中忽视这些活动将会导致各项观测指标出现明显的误差。例如事务处理工作负载有 20% ~ 30% 的时间是执行在操作系统模式下。此外，应用在操作系统模式下的行为与在用户模式下的行为是完全不同的，因此，对商业应用来说，真实模拟所有系统组件是十分必要的。

图 9-1 从概念上给出了用户级模拟器与全系统模拟器的区别。从图中可以看出，全系统模拟器可以运行目标操作系统，且在该操作系统上，应用程序可以在模拟的目标机器上直接执行系统调用。而在用户级模拟器中，基准测试程序发起的所有系统服务都需要绕过模拟器，而由底层宿主机的操作系统来处理。

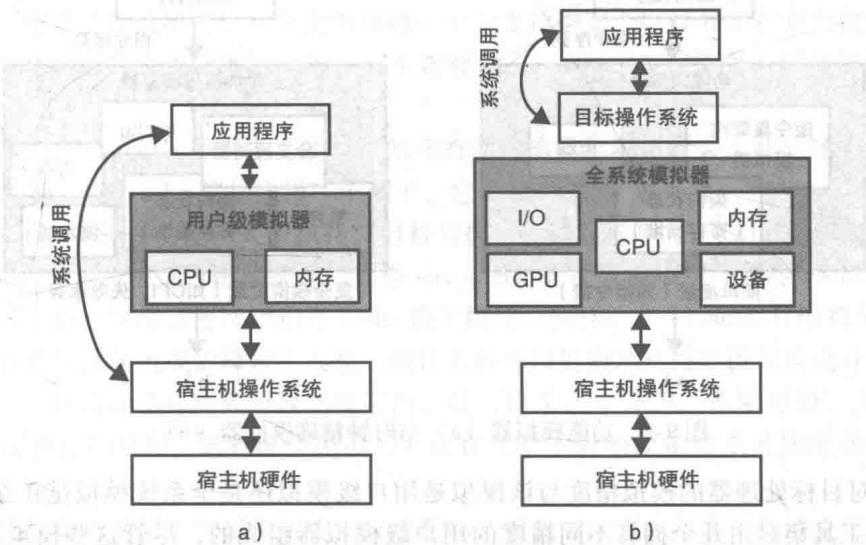


图 9-1 用户级模拟器 (a) 与全系统模拟器 (b)

现在已经有很多基于不同设计目的的用户级模拟器，比如 SimpleScalar、Asim、MINT、RSIM、Zesto、Shade 等，这些工具用来评估诸如 IPC、分支预测准确度、cache 命中率等微架构性能指标。SimpleScalar 由于其灵活性和易于获取而被广泛使用。从历史上看，性能一直是体系结构评估中最重要的评估指标，然而近年来，人们开始把更多的研究兴趣放在功耗和能量上，已经有一些用户级的功耗和能量评估工具开发出来，例如，Wattch 就是在 SimpleScalar 的基础上增加了体系结构级别功耗评估的扩展，它会在每个周期调用各活动单元的功耗模型以计算并记录所消耗的功耗，其他体系结构级的功耗评估工具还有 SimplePower 和 TEMPEST。

9.2.2 功能模拟器和时钟精确模拟器

本小节介绍一种基于模拟的细节程度进行正交分类的方法。最简单的模拟器仅仅对功能进行模拟，比如只模拟处理器中每条指令的功能，而不考虑任何微架构上的细节模拟，这种类型的模拟器通常用于构建更复杂的模拟器，在对目标处理器进行详细微架构模拟之前，其指令集架构 (ISA) 就是通过这种模拟器正确模拟的。这种方法将目标处理器的指令集逻辑/功能建模和微架构建模进行了拆分。

时钟精确模拟器可以模拟所有微架构模块的细节，它们不仅仿真了各种微架构模块的功

能,通常还会跟踪记录对应的时间,正是由于可以对时间进行跟踪记录,这类模拟器可以提供性能结果,因此可以用来对不同设计选择进行评估。为了使模拟器操作正确,模拟器设计开发人员必须对微架构的设计进行详细编码,并为模拟器提供时序输入信息。为了快速地评估设计空间,模拟器中的大多数微架构模块都是参数化的,例如,cache 组件的大小、相联度以及块大小等,这些参数都可以配置为用户定义的值,从而允许设计空间探索各种不同的 cache 配置而无需重新编译模拟器代码。大多数模拟器都使用一个专门的目标处理器配置文件作为模拟器的输入,基准测试程序也在目标机上执行。

图 9-2 显示了功能模拟器和时钟精确模拟器的区别。由于功能模拟器不对任何微架构缓冲区进行模拟,因此它比时钟精确模拟器要快得多。功能模拟器不深入模拟目标系统,它可以提供一些简单的统计信息,例如指令数或静态/动态指令比例等。与之相反,时钟精确模拟器则提供了更多细节信息,更有助于发现目标机的性能瓶颈。

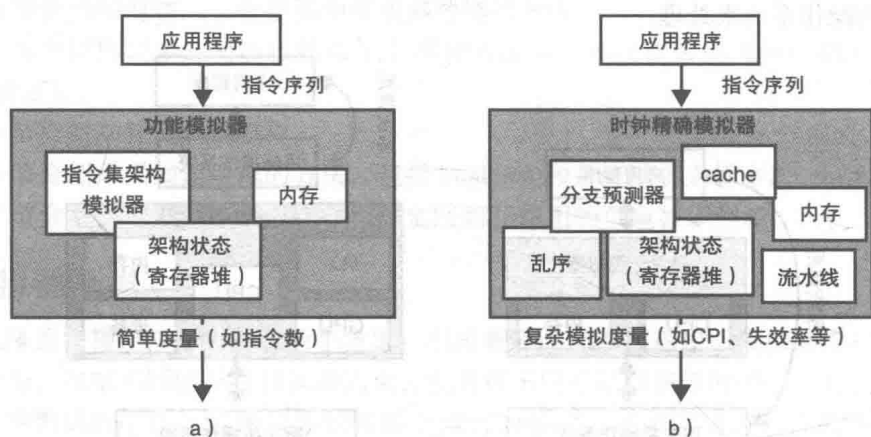


图 9-2 功能模拟器 (a) 与时钟精确模拟器 (b)

模拟器对目标处理器的模拟精度与该模拟是用户级模拟还是全系统模拟是正交的。例如, SimpleScalar 工具集是由几个拥有不同精度的用户级模拟器组成的, 尽管这些模拟器共享同一个指令集, 但它们在模拟粒度和速度方面却有不同的侧重。在 SimpleScalar 中, 微架构建模与功能模拟是相互分离的, 功能的正确性通过一个机器定义文件来提供, 该文件定义了目标指令集架构 (ISA) 中每条指令的功能行为。例如, 机器定义文件指定一个加法 (ADD) 指令有两个寄存器操作数, 并在两个寄存器内容相加后用结果更新目标寄存器。如果 ISA 中的某条指令影响控制寄存器, 如条件码寄存器, 也可以在机器定义文件中指定。机器定义文件在 sim-fast 中使用, sim-fast 是 SimpleScalar 模拟集中速度最快、粒度最粗的一个, 它是一种指令级模拟器, 在不考虑微架构细节的情况下模拟指令的执行。sim-fast 只跟踪目标机的系统结构状态, 例如寄存器堆内容等。模拟器读取基准测试程序中执行的每条指令, 并基于机器定义文件规范来描述指令执行后对体系结构的影响。

在 SimpleScalar 模拟集中的另一个模拟器是 sim-outorder, 它是 SimpleScalar 工具集中模拟粒度最细的一个, 可以模拟多级 cache 层次、分支预测、乱序发射以及时序精确的执行。该模拟器像 sim-fast 一样读取每条指令并完成指令的功能执行, 然后调用目标处理器的一个具体微架构模型来计算时间。例如, 当执行 load 指令时, 功能模拟器首先访问目标存储, 并将正确的数据存入目标寄存器, 然后时钟精确模拟器模拟整个内存层次结构, 首先访问 L1 cache, L1 cache 模拟模块只记录地址标签 (tag) 信息, 而不保存真实数据, 真实数据是在功能模拟阶段直接从模拟内存中访问的。若 L1 cache 失效, 即访存地址与任何 L1 cache 的地址标签都不匹

配,则继续往下访问 L2 cache,最终所要访问的数据可能需要到内存中才命中。模拟器也会对执行时间进行跟踪记录,这个时间包括访问 L1 cache 的周期数、访问 L2 cache 的周期数以及最终访问内存的周期数。除了 cache 访问外, sim-outorder 还考虑了资源约束的影响,例如, cache 是否是非阻塞的、层次化的 cache 中每个层次的读端口数量等。在这种模拟器中,模拟详细程度的不同主要是取决于我们在微架构模型设计和开发工作中所付出的努力。

微架构配置参数定义了目标处理器内每个主要部件的具体结构,其中比较重要的参数包括:取指宽度,发射宽度,退出宽度,分支预测器的类型和大小, cache 级数,每级 cache 的配置与大小,功能单元的数量,重排序缓冲区的大小。其中一些参数只改变结构的大小或延迟,而另一些参数改变微架构部件的功能行为。在这种情况下,实现设计人员评估时感兴趣的那些微架构功能是十分必要的。默认情况下, sim-outorder 实现了多个可供替换的微架构块组件,其中分支预测器就是一个很好的例子,它的每个预测器都有不同的功能行为。例如, SimpleScalar 中提供了不同的分支预测器供选择,包括 2-bit 分支预测器、两级分支预测器以及混合分支预测器等。配置文件来决定运行时实例化哪一个分支预测器,以及分支预测器的大小。如果设计人员对评估新的分支预测器感兴趣,那么需要在 sim-outorder 源代码中添加新预测器对应的模拟代码。

sim-fast 和 sim-outorder 分别是功能模拟器和时钟精确模拟器的代表,此外还有一些模拟方法介于二者之间。sim-cache 就是这样一个例子,它由 sim-fast 扩展而来。在 sim-cache 中,每条指令都像 sim-fast 一样在功能模式中执行,但每当执行到访存指令时, cache 模拟模块就会被调用,它对 cache 是否命中进行跟踪统计。如果 sim-cache 配置为模拟一个指令 cache,那么目标处理器取指每条指令时都需要调用指令 cache 模拟模块。同样, sim-branch 只模拟分支预测器。借助于这些粗粒度模块组成的模拟工具集,设计人员可以更快地在特定微架构设计空间中完成探索和筛选,一旦设计选择降到可控的范围内,就可以采用更详细、速度更慢、类似 sim-outorder 这样的时钟精确模拟。这种混合模拟的方法在几乎不影响模拟结果准确度和精度的前提下提高了模拟速度。

全系统模拟器考虑了它与操作系统交互的影响,但它可能没有对微架构的全部细节进行模拟。例如, SimOS 是一个全系统模拟器,它在指令集级别对应用程序执行与操作交互进行了模拟,在这个抽象级别,模拟速度足够快,因此可以通过硬件性能计数器来测量运行在真实机器上的实际大小工作负载的执行统计数据。这种类型的模拟器在以调试为目的的系统软件开发过程中也是很有价值的,它为系统软件开发人员提供了一个在速度和硬件细节之间进行平衡的良好方法。另一个大家所熟知的全系统模拟器是 Simics, Simics 是一个工业级的全系统模拟器,它可以启动不加修改的操作系统, Simics 的底层处理器模型可以支持多种指令集架构,例如 Sparc、x86 和 IA-64 等,它几乎实现了所有的功能指令,甚至包括那些在大多数基准测试程序中很少使用的指令,因此,它可以启动不加修改的操作系统,并且可以执行任何需要全系统支持的复杂工作负载。

全系统模拟器可以结合乱序微架构模型来增强其功能。例如, PHARMSim 将 SimOS 和 SimpleMP 进行了结合,是 SimpleScalar 的多处理器扩展。另一个此类实验工具是 SoftWatt,它在 SimOS 的基础上,通过增加 MIPS 微架构模型来进行功耗预测。

9.2.3 trace 驱动模拟器、执行驱动模拟器和直接执行模拟器

处理器模拟器需要对运行在目标机上的基准测试程序的指令执行过程进行模拟。有两种获取指令来驱动模拟器的方式: trace 驱动模拟和执行驱动模拟。

trace 驱动模拟

在 trace 驱动模拟中, 首先基准测试程序在指令集架构兼容的处理器 (或模拟器) 上执行, 这个兼容处理器不一定要和目标机完全一样。当基准测试程序执行时, 把每条指令记录到一个 trace 文件, 这是通过实际机器上的硬件监控或模拟器 (这种模拟器也称为 trace 生成器) 来实现的。收集的 trace 信息可能是基准测试程序的完整执行过程, 也可能是基准测试程序中设计人员感兴趣的一小部分, 并且通常后者更为常见。在开始 trace 收集前, 处理器的整个架构状态记录到 trace 文件中, 在 trace 收集阶段, 可以使用一些 trace 收集的基础工具 (例如用户级软件 trace 生成器) 来记录系统调用, 在这种情况下, trace 生成器可能仅仅记录系统调用执行前后的系统架构状态。与之类似, 当记录 trace 过程中遇到中断事件时, trace 生成器可以记录中断前后的系统架构状态。因此, 每个 trace 文件就是一系列 trace 记录的集合, 再加上一些系统架构状态信息。在多处理器或 CMP 上的执行 trace 中, 来自不同线程的 trace 记录交叉存储在同一个 trace 文件中, 其存储次序与它们在 trace 生成器中执行时所发生的次序一致。

一旦收集完成, trace 就被输入到时钟精确模拟器, 并在模拟器中使用详细的微架构模型对来自 trace 文件的每条指令进行模拟。时钟精确模拟器和用户级模拟器首先载入 trace 文件中与系统架构状态相关的信息, 然后开始详细执行每条指令, 当时钟精确模拟器遇到无法模拟的系统调用或中断时, 它只是简单地从 trace 文件中读取系统调用之后的系统架构状态, 并更新自己的状态内容, 这样就模拟了目标机上系统调用在系统架构上的影响。

trace 驱动模拟的一个缺点是, 这种模式下时钟精确模拟器无法准确量化分支预测的影响。trace 驱动模拟器无法模拟分支预测错误路径, 因为 trace 信息本身就不包括任务错误路径上的指令。因此, trace 驱动的时钟精确模拟器只能简单地加上一个分支预测失败开销, 而不是真正地对错误路径进行模拟。此外, 在模拟大规模工作负载时, trace 文件的大小也是一个需要重点关注的问题。

另一方面, trace 驱动模拟器也有很多优点。首先, 它只需要收集一次 trace, 就可以重复利用收集到的信息对各种微架构配置进行模拟。此外, 也可以让 trace 生成器在 trace 收集的同时记录一些体系结构信息, 从而简化时钟精确模拟器的设计。在记录 trace 时, 当碰到目标架构下的某条指令需要与多个微架构组件进行复杂交互时, trace 生成器可以将执行完该指令后对系统架构的影响记录下来, 这样, 在后续模拟中, 当时钟精确模拟器碰到这种复杂指令时, 并不需要完全真实地再现该指令执行的过程, 而是可以简单忽略掉, 但是通过读取记录该指令对系统架构影响的 trace 文件来更新自己的系统架构状态。最后, trace 文件可以是任何模拟器组件的一个固定输入, 这意味着即使我们所研究的目标系统结构特征发生了变化, 驱动模拟器执行的事件序列也始终不变, 并且对特征的评估不依赖于目标系统架构中其他部分的行为。这一点在多处理器系统模拟中也是非常重要的, 多处理器中来自不同线程指令动态交错执行可能会影响最终的行为, 不同 cache 协议特性的比较就是一个例子。

目前已经有很多基于 trace 驱动模拟技术的变体, 比较常见的一个方法是在 trace 收集期间只记录访存的地址, 这种 trace 也叫作内存访问流, 收集好后的内存访问流作为 cache 模拟器的输入, 这种 cache 模拟器只模拟微架构特征, 而不模拟具体的数据。在使用 trace 信息来研究存储系统的时候, trace 文件只包含基准测试程序的内存访问流, 而不保存目标系统架构状态。

执行驱动模拟

执行驱动模拟器不依赖于 trace 文件, 而是以基准测试程序作为模拟器的输入。执行驱动模拟器通过解析基准测试程序执行文件来加载系统架构状态, 然后在目标机上模拟基准测试程序的执行。除了真实再现目标结构的功能之外, 还必须再现对应的执行时间。这类模拟器必须知道如何处理指令集中每条指令的执行时间和具体功能, 哪怕这条指令在基准测试程序的执行

过程中只出现一次。由于执行驱动模拟器将时序和功能集成在一起,因此开发起来比较复杂,但是执行驱动模拟器在模拟精度、灵活性以及真实性等方面都比 trace 驱动模拟器要好。

直接执行模拟

trace 驱动模拟器和执行驱动模拟器在宿主机上对目标系统的每条指令都进行了模拟,这大大降低了模拟速度。而直接执行模拟很好地平衡了模拟精度和速度两个方面,它在宿主机硬件上直接执行基准测试程序中的指令来兼顾模拟精度和速度。当指令直接在宿主机硬件上执行时,模拟器不提供任何时序相关的数据。因此牺牲了一定的精确度。设计师们要确定哪些目标系统组件需要详细模拟,例如,如果设计师只对模拟数据 cache 的性能感兴趣,那么除 load/store 指令外的所有指令都可以直接在目标硬件上执行,当基准测试程序中遇到 load/store 指令时,会调用存储层次模拟器来详细模拟存储层次的性能。直接执行模拟中,要求目标指令集架构必须和宿主机的 ISA 相同,这样基于目标体系结构编译的基准测试程序才能在宿主机硬件上运行。

在直接执行模拟中,基准测试程序的一部分代码在宿主机上执行,另一部分代码在模拟环境中执行。要实现这种复杂的交互,需要对基准测试程序在源码级或者目标代码级进行插桩,当基准测试程序执行到感兴趣的事件时,就会调用相应的模拟模块进行更详细的模拟。我们可以借助编译器对基准测试程序代码进行分析,并识别出所有感兴趣事件的位置。比如,为了评估数据 cache 的性能,编译器需要识别出所有的 load/store 指令,然后在每条 load/store 指令之前插入对模拟模块的函数调用,这些模拟模块可以将访存指令的有效地址、程序计数器以及任何其他评估相关数据作为输入。这样,当插桩过的二进制程序在宿主机上执行时,每当执行到 load/store 指令,就会调用相应的模拟模块执行。

插桩可以在源码级完成,在没有源代码时,也可以使用二进制重写工具对基准测试程序的二进制文件进行插桩。现有一些流行的二进制重写工具,比如针对 Alpha 指令集架构的 ATOM 就是一个,下面的例子展示的是一小段 ATOM 的插桩代码,简单说明了 ATOM 如何重写二进制文件,以使其能够在宿主机上直接模拟执行。

```
Instrument (int argc, char ** argv, Obj *obj)
{
    Proc *p, Block *b, Inst *inst;
    for (p = GetFirstObjProc (obj); p != NULL; p = GetNextProc (p)) {
        for (b = GetFirstBlock (p); b != NULL; b = GetNextBlkc(b)) {
            for (Inst = GetFirstInst (b); inst != NULL; i = GetNextInst(Inst)) {
                if ((IsInstType (i, InstTypeLoad) || (IsInstType (i, InstTypeStore))) {
                    AddCallInst (Inst, InstBefore, "SimulateCache", EffAddrValue);
                }
            }
        }
    }
}
```

ATOM 工具将基准测试程序二进制文件作为输入,并使用上面代码片段中所示的过程生成插桩代码。第一层循环遍历基准测试程序二进制中的每一个过程,第二层循环遍历给定过程中的每一个基本块,第三层(最内层)循环遍历基本块中的每条指令,因此,这三层循环遍历了基准测试程序中的每一条指令。上述代码片段中的 if 条件语句对指令是否是 load 或 store 指令进行检查,若是 load 或 store 指令,ATOM 会在原始基准测试程序中的内存访问指令之前,插入一个名为“SimulateCache”的函数调用,SimulateCache 函数需要内存访问指令的有效地址作为输入参数。ATOM 实际上生成了一个新的二进制文件,里面的每次访存操作都会调用 Sim-

ulateCache 函数。SimulateCache 函数可以是用户定义的任何 cache 模拟程序，可以量化不同 cache 配置对基准测试程序性能的影响。当插桩过的基准测试程序在宿主机上直接执行时，cache 模拟函数也同步执行，两者结合构成了一个直接执行模拟的形式。ATOM 提供了一系列 API 用来探测各种架构状态，例如，用户可以在基准测试程序执行过程中的任意时刻检查指令的类型（比如是 load 还是 store），读取寄存器值或内存内容。

另一个插桩工具的例子是 PIN，这是一个基于 x86 和 IA64 指令集的二进制插桩工具。ATOM 通过对二进制插桩的方式离线生成一个新的二进制文件，因此插桩后的二进制可以直接在宿主机硬件上运行。与之相反，PIN 使用的是即时编译（just-in-time, JIT）方法，和 ATOM 一样，PIN 的输入是基准测试程序二进制以及插桩程序，PIN 的运行时环境能够截获程序中每个基本块的第一条指令，然后为二进制文件中的每个基本块自动在线生成新的代码，并写入内存的代码 cache 中。此后，处理器直接从代码 cache 中取指令执行，而完全不会感知到原始基准测试程序二进制文件的存在。内存中的代码 cache 由 PIN 进行软件管理，当遇到在之前执行阶段翻译过的基本块时，PIN 可以重用代码 cache 中的代码。当基本块正常执行、没有进行模拟代码插桩时，基本块对应的 JIT 代码与原始的二进制代码一样。

通过二进制代码插桩实现的直接执行也有一些额外开销。当 ATOM 进行二进制代码插桩时，插桩代码本身执行时也需要占用一定的寄存器，这样，ATOM 必须保存好插桩代码使用的所有寄存器，当插桩代码退出时再将其恢复，因此，直接执行期间会产生大量的寄存器分配开销。而在 PIN 中，代码插桩是在线完成的，它可以对代码进行分析从而获知在代码插桩前，哪些寄存器是活跃的，哪些是挂起的，JIT 编译器先选择那些挂起的寄存器给插桩代码使用。动态插桩本质上提供了更多的运行时可见性，因此可以更好地实现代码优化；另一方面，PIN 采用动态插桩的方法，使得对同一份代码序列，可能产生多次 JIT 编译的开销，而 ATOM 每次重编译时只有一次编译开销，并且没有额外的运行时编译开销。

9.3 模拟器的集成

根据前面的介绍可以看出，功能模拟器是最容易实现的，但它在目标设计的性能瓶颈分析中只能提供非常有限的作用。而执行驱动模拟器由于需要正确模拟执行的功能和时序，因此开发工作量很大。通过精心规划，集成现有的工具是加速新模拟器开发的有效方法，这种方法可以显著提高现有工具的模拟能力，并避免从头开始开发一个新的模拟器。特别是当现有的工具之间功能互补时，对它们的集成就更有意义。一种常见的模拟器集成方法是时钟精确模拟器和全系统模拟器集成在一起。

9.3.1 功能优先模拟器的集成

我们之前介绍过，时钟精确模拟器是对系统微架构的细节进行模拟，而全系统模拟器则是在功能级别模拟整个系统。为了获得可以对系统微架构细节以及时序信息进行模拟的全系统模拟器，方法之一就是集成上述两类模拟器，这种集成方法称为功能优先模拟。在该方法中，全系统功能模拟器上运行基准测试程序，并生成指令序列，功能模拟器维护目标机架构状态的一个副本，比如寄存器文件的拷贝，在每条指令执行后，更新其架构状态，然后指令输入给模拟器的时序部分，时序部分可以执行系统微架构模拟并记录时序信息。时序模拟器也需要维护一个架构状态的副本，用于其自身的时钟精确模拟。

时序模拟器对流水线和诸如 cache、分支预测器这样的微架构进行详细模拟，从功能模拟器接收到的每条指令，时序模拟器都会模拟其执行过程，并记录指令在流水线中由于控制相关、数据相关和结构相关所消耗的时间。时序模拟器还负责所有与预测相关行为的模拟，比

如,当遇到分支指令时,如果目标处理器中有分支预测器,则时序模拟器访问分支预测模块,并预测分支指令的跳转方向和目标。在功能优先模拟方法中,功能模拟器先执行,并将提交的指令输入给时序模拟器,只要时序模拟器在正确的路径上执行,功能模拟器就一直提供正确路径上的指令。在控制指令执行后,时序模拟器可能偶尔会由分支预测器定向到错误的执行路径上,在这种情况下,时序模拟器可能只知道分支方向和分支目标地址,但并不访问错误路径上的指令。当碰到控制相关时,功能模拟器不会出现预测错误的情况,因为功能模拟每次执行一条指令,分支指令的结果可以立刻知道。为了模拟错误路径执行,时序模拟器需直接访问功能模拟器中的存储模型,以便执行和恢复在错误路径上的指令。

功能优先模拟器集成的一个成功例子是 SimWattch,它集成了 Simics 和 Wattch,前者是工业标准产品,后者是学术版的 SimpleScalar,可以统计性能和功耗。

9.3.2 时序优先模拟器的集成

一种替代功能优先模拟器集成的方法是时序优先模拟,在这种方法中,时序模拟器的运行优先于功能模拟器。时序模拟器模拟了目标系统微架构配置,因此可以在周期级别模拟指令的执行。时序模拟器不必执行所有的指令,特别是,指令集中可能存在一些在实际基准测试程序中很少执行到,但需要进行复杂系统微架构交互的指令,如果我们可以忽略对这类很少使用指令所需的复杂微架构交互的模拟,那么时序模拟器的发展和验证就能大大简化。在时序优先模拟器中,时序模拟器在指令即将提交时必须就指令执行结果和功能模拟器进行验证,主要是验证时序模拟器和功能模拟器的系统架构状态是否相同。

时序模拟器通过功能模拟器来获取指令执行对系统架构状态的影响。由于功能模拟器总是如实地记录系统架构状态的改变,因此,时序模拟器将功能模拟器的系统架构状态视为对比标准。如果时序模拟器的系统架构状态与功能模拟器的状态相匹配,则时序模拟器正常退出该指令。这种能够通过退出检查的指令被称为正确执行的,否则被称为状态偏离的,一旦出现状态偏离,就将功能模拟器的系统结构状态复制到时序模拟器中,时序模拟器将在出现状态偏离的指令之后立即清空流水线,并重新获取后续指令执行。因此,即使时序模拟器不能正确产生指令对系统架构状态的影响,它也还是可以继续执行下去。需要注意的是,此时刷流水线的操作是模拟器自身引起的,因此,流水线恢复的过程将对系统时序模拟造成一定的影响,如果恢复操作不是频繁发生的,那么这个影响就很小。在时序优先模拟器中,时序模拟器不会更改功能模拟器的系统架构状态,因此,时序模拟器决定指令的调度执行顺序,但最终的执行状态结果是由功能模拟器决定的。

GEMS 和 FeS2 是时序优先全系统模拟器的两个实例,它们分别对 SPARC 指令集和 x86 指令集进行了模拟。GSMS 和 FeS2 都使用了 Simics 作为其功能模拟器,时序模型则是 GEMS 框架中的一个新组件。根据所需的精度级别,GEMS 可以仅模拟作为时序模型中的存储层次结构,或者也可以在时序模块中模拟一个详细的乱序处理器。GEMS 的存储层次结构模块称为 Ruby,它可以模拟目标微处理器的复杂存储层次结构,包括片上互连网络、DRAM 控制器和内存条等;处理器模块称为 Opal,它可以实现一个详细的乱序处理器模型。GEMS 可以通过配置方式来启动 Ruby 或 Opal,当两者都启动时,则可以模拟详细的内存模型和处理器模型。由于 GEMS 是一个时序优先模拟器,时序模型的运行优先于功能模型(Simics),例如,Opal 模拟指令执行时,将其送入对应的处理器流水线中进行模拟,当指令准备提交时,它再调用 Simics 功能模拟器执行对应指令,然后通过比较 Opal 与 Simics 的系统架构状态来检查 Opal 执行功能的正确性,如果 Opal 的指令执行结果是一致的,则直接退出该指令,否则,如果 Opal 的指令执行结果不一致,则需要从 Simics 中重新加载正确的系统架构状态。本质上,Opal 控制 Simics 何

时可以执行下一条指令，从而也就控制了每条指令执行的时序。

在某些情况下，时序模拟器对功能模拟器模拟进度的控制是十分关键的，特别是模拟多处理器目标系统时，不同线程指令的交叉执行顺序是依赖于时序信息的，此时，时序优先的方法比功能优先的方法更加准确。

9.4 多处理器模拟器

到目前为止，对模拟器的讨论还主要集中在对单处理器的模拟，随着 CMP 的出现，并行系统开始广泛用于从服务器到手机的各个计算领域。工业设计人员也更加关注能够支持科学计算、Web 服务、数据挖掘和计算金融等高性能应用的 CMP 系统的设计。每一代新的 CMP 系统通常都会使用更加复杂的互连结构来连接更多的处理器核，因此相比前代都会变得更加复杂，这导致对 CMP 系统的模拟成为一个新的挑战。构建复杂的 CMP 系统模拟平台变得十分困难，过程也非常耗时。模拟单处理器目标机和多处理器目标机的一个根本区别是，在多处理器目标机中，不同的目标处理器核之间存在交互和影响，一个处理器核的行为可能会影响另一个处理器核的执行，这种交互可能出现在片上互连网络或者目标线程之间进行数据交换时，例如，基于总线的 CMP 中的多个处理器核需要共享总线上的请求，而这些请求之间可能会互相冲突，在缓存一致性相关的事务请求中也会出现类似的交互，准确模拟这些交互行为使得多处理器模拟比单处理器模拟更具挑战性。

目前有两种方法来模拟 CMP 系统：串行模拟和并行模拟，如图 9-3 所示。

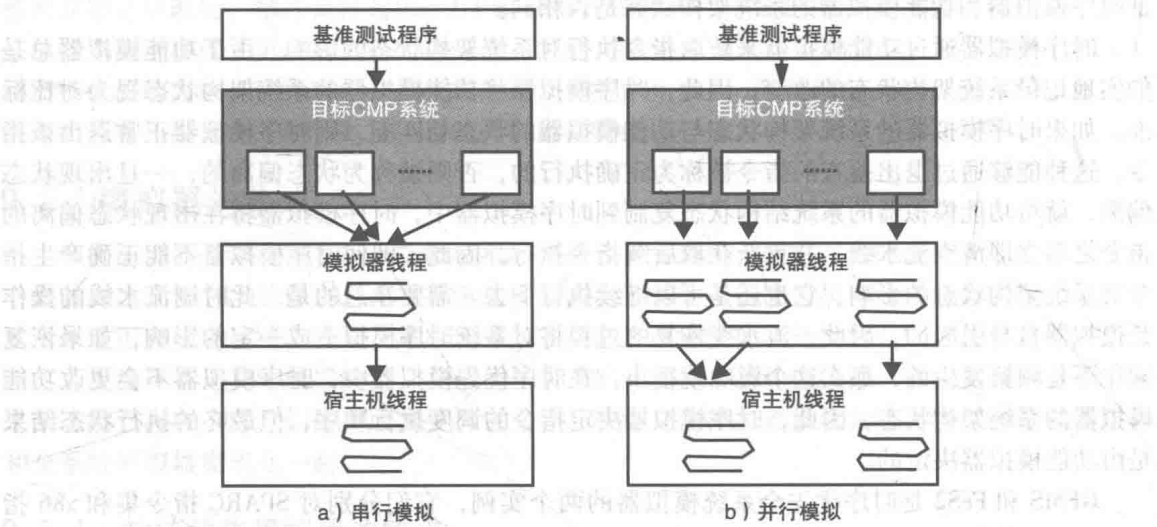


图 9-3 并行目标机的模拟

9.4.1 串行多处理器模拟器

如图 9-3a 所示，在多处理器目标机的串行模拟中，所有的目标处理器核都在一个模拟器线程中执行。这种模拟器通常以轮询的方式依次模拟不同处理器核的时钟周期，比如，先模拟目标处理器 core 0 的一个周期，紧接着再模拟目标处理器 core 1 的一个周期，以此类推。周期是模拟的一个基本时间单位，在每个周期中，模拟器轮流模拟每个处理器核的行为，相应地将流水线推进一个时钟周期，此外还模拟处理器核的私有 cache 的行为。完成一个处理器核的模拟后，在同一周期中，紧接着模拟下一个处理器核的行为，一旦目标系统中所有处理器核都完成了同一周期的模拟，该模拟器就接着模拟本周期中发生的全局事件，包括：某个处理器核私

有 cache 中发生的 cache 失效, 该请求被送入最后一级共享 cache, 或者转换成一个核间通信事件, 即数据通过片上互连网络从一个处理器核传送到另一个处理器核。当全局事件模拟完成时, 模拟器就进入下一个模拟周期, 此后一直重复这个过程, 直到整个模拟结束。在每个周期的最后进行全局事件的处理, 可以保证一个目标处理器行为的预期效果在其他处理器核中准确反映, 因此, 串行模拟多核能够真实模拟处理器核间的交互。不过, 由于目标系统的处理器核数越来越多, 而单线程的性能却没有太大提高, 导致串行模拟器的性能难以满足灵活扩展的需求。

现有的串行多处理器模拟工具包括 RSIM、M5、MINT、GEMS、SESC 和 Zesto 等。此外, 诸如 Simics 和 SimOS 这样的全系统模拟器也能够模拟多处理器系统。RSIM 是第一个广泛使用的能够模拟乱序执行处理器核的多处理器模拟器, 它结合了指令级并行 (ILP) 和多处理器模拟, 但它无法运行复杂的工作负载, 如操作系统代码等。近年来, 人们开发了一些能够运行商业负载、具有详细系统微架构特征的多处理器模拟器: M5 是一个面向网络模拟的全系统模拟器, 它支持详细的乱序 SMT 处理器模型; SESC 能模拟不同的处理器架构, 包括单处理器和片上多核处理器, 它支持对现代处理器中带有分支预测和 cache 结构的乱序流水线以及其他部件进行模拟; GEMS 则将功能模拟和时序模拟进行了分离, 它可以使用一个叫作 Ruby 的存储层次模拟模块对多处理器的存储层次结构和 cache 一致性协议进行模拟, GEMS 还包含一个能够配置成多处理器时序模拟器的 Opal 模块, 可以对多处理器中的每个处理器进行详细模拟, 借助于 Opal 和 Ruby 模块, 设计人员可以模拟处理器核、cache 层次结构, 以及多处理器环境下的 cache 一致性协议等。

RPPT (The Rice Parallel Processing Testbed) 是最早的多处理器模拟器之一, 它通过在宿主机硬件上直接执行基准测试程序代码, 以及选择性地模拟处理器间的通信事件来实现模拟加速。RPPT 的输入参数主要有三个, 分别是: 目标处理器结构参数, 互连网络配置, 多线程基准测试程序。RPPT 使用编译器分析或用户提供的注释来识别基准测试程序代码中不同线程的交互点, 它侧重于模拟多处理器目标机上的消息传递。在同一线程中执行的局部代码被划分成基本块, 通过对目标处理器的配置来估计每个基本块执行所需的周期数, 如果一个基本块由一个加、一个减、一个乘以及一个除组成, 则 RPPT 将每个独立指令的延迟进行求和。对 load 指令来说, 其延迟根据预期的 cache 命中率以及命中和未命中的延迟来近似计算得到。简单的周期计数功能可以通过在每个基本块的末尾插桩代码来实现, 无论线程在何处与另一线程进行通信, 功能调用都可以插入其中, 以此实现对目标机互连网络的模拟, 插桩的基准测试程序将在宿主机处理器上运行。

9.4.2 并行多处理器模拟器

传统单线程模拟扩展性不佳, 因此人们又对并行多处理器模拟器进行了深入研究。尽管并行多处理器模拟器实现难度大, 但它可以显著加快模拟速度。在这种方法中, 每个目标处理器核都在单独的模拟器线程上进行模拟, 多个模拟器线程被映射到一个宿主机处理器核上。假设 C 为目标 CMP 中的处理器核数目, N 为整个宿主机 CMP 中硬件线程的数量 (可能包含多线程处理器核), 一种直观且可扩展的模拟负载分配方法是, 将同一目标处理器核的模拟分配到一个模拟线程, 然后在宿主机中建立 C/N 个模拟线程到每个硬件线程的映射。无论从性能角度还是编程角度, 这种方法都是可扩展的。图 9-3b 阐述了并行 CMP 模拟的概念。

并行 CMP 模拟的速度主要依赖于以下 4 层的实现效率: 应用程序层, 目标硬件层, 宿主机硬件层, 模拟层。这 4 层的详细阐述见图 9-4。在应用程序层中, 模拟加速受限于算法加速, 如果目标应用只有很少并行性或根本无并行性, 则通过在 CMP 上运行模拟得到的加速是微乎

其微的，这是因为每个宿主机线程的上下文只是一个模拟器的目标处理器核。如果目标 CMP 架构本身存在一定的性能瓶颈，如处理器核数少或者存储架构效率较低，则并行模拟对模拟加速没有什么帮助。例如，如果目标处理器核大多数时间都在等待内存，则模拟目标处理器核的模拟线程也大多处于空闲状态。在宿主机硬件层，CMP 宿主机必须有足够的资源，这样才能够有效执行并行模拟，例如，通过支持快速读写共享和拥有足够的片上 cache 来维护模拟的工作集。

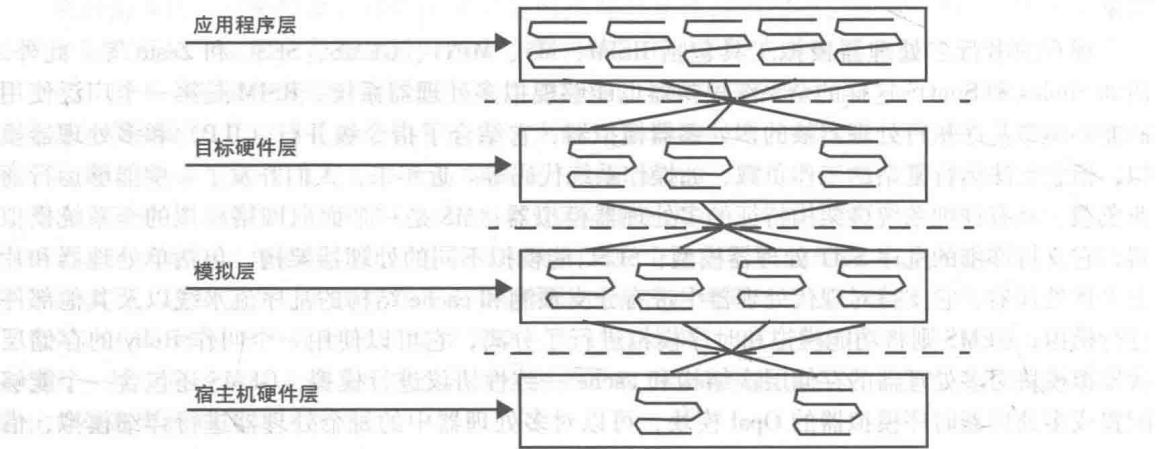


图 9-4 并行 CMP 模拟的 4 层

在本节中，主要讲述模拟层次的性能。首先讨论一个更为普遍的问题：模拟处理器核的宿主机线程之间的交互以及它们之间的过度同步。而基准测试程序中固有的低效现象、目标硬件层和宿主机硬件层等不在本节的讨论范围之内，目标 CMP 设计的相关问题在本书的其他部分进行论述，而有效的并行算法设计则完全不在本书的主题范围内。

为简化起见，我们假设宿主机系统的处理器核数与目标处理器核数相等，在这种情况下，每一个目标处理器核都在一个宿主机核上进行模拟，这种方法以设计和开发模拟环境的复杂度为代价，换取了模拟速度的提升。当多个目标处理器核并行地运行在多个宿主机核上时，模拟器必须协调各模拟线程，以便处理器核间的交互能够正确地模拟。由于不同的目标核在不同的宿主机核上以不同的速度进行模拟，因此要满足这一要求是一个极大的挑战，存在的主要问题是，一个处理器核的模拟可能在另一个处理器核的模拟之后完成，当发生这种情况时，执行靠前的处理器核模拟线程可能会因为与后面处理器核模拟线程的交互而受到影响，因为这一交互行为本来是需要在执行更快线程的过去某个时间的，但是现在才收到，所以会对当前状态产生影响。在这种情况下，因果关系的处理顺序与它们在目标模拟系统中的时间顺序是不一致的，出现了模拟准确性上的偏离。

在深入研究 CMP 的并行模拟之前，我们对前人已经探索过的并行离散事件模拟（Parallel Discrete-Event Simulation, PDES）环境下的各种技术进行概述。

并行离散事件模拟

多年来，并行模拟一直是众多应用领域中的热门研究话题。在用于计算机体系结构模拟之前，并行化就是一种非常流行的用来加速离散事件模拟（DES）的方法。离散事件模拟所针对的是不连续变化系统的模拟，即变化经常发生在离散的时间，比如某个特定事件发生的时候。并行离散事件模拟（PDES）采用两种不同的机制来实现模拟中因果关系的正确有序：保守同步机制和乐观同步机制。

保守同步机制的关键是通过仔细地逐步模拟来避免时序的冲突，这可能会降低模拟的效

率。为了避免时序的冲突,一个保守的方法是,仅当没有其他事件可能对当前事件产生影响时再处理该事件,换句话说,如果事件 A 对事件 B 有影响,则保守方法会先处理事件 A,然后再处理事件 B。最严格的保守同步机制是按照目标机中事件发生的确切顺序来模拟事件,这种方法实际上并没有实现并行。

基于前看 (lookahead) 的方法是一种更为宽松的保守同步机制,lookahead 值是指某个模拟事件在不违背时序正确性的情况下可以安全执行的最大时间量。在这种方法下,先发生的事件总是先被处理,并且允许模拟线程从该事件发生时刻开始,向前直接模拟推进 lookahead 的时间。

对 PDES 来说,最流行的保守同步机制是 barrier 同步 (barrier synchronization),它将模拟过程划分成由若干时间单元组成的时间间隔,这些时间间隔通过同步 barrier 进行分离。在一个时间间隔内,所有的模拟线程在到达同步 barrier 前,都可以独立推进执行。该方法要求在同一时间间隔内,事件之间不会相互影响。当模拟进入下一个时间间隔之前,在当前时间间隔内触发的所有全局事件必须对其他模拟线程可见,这样它们产生的效果才能在下一个时间间隔中进行模拟。

与保守的 PDES 方法相比,乐观的 PDES 方法在模拟加速的同时可能会产生因果冲突。即使在目标机中事件 A 可能会影响事件 B,且事件 B 应该发生在事件 A 之后,在乐观模拟方法中,事件 B 的模拟也有可能在事件 A 的模拟之前进行。如果后面发现事件 A 确实影响了事件 B,则模拟过程必须进行回滚,从而恢复原有的因果关系。乐观同步机制需要利用模拟状态的周期性检查点,并且需要检测时间冲突。当检测到时间冲突时,模拟器必须回滚到距离当前最近的检查点,并且从这个检查点以一种安全模式重新执行模拟过程。

当时间冲突发生频繁时,乐观同步机制的性能很差;当时间冲突很少发生时,保守同步机制的性能更差。

因为多处理器的处理器核受到时钟约束,其状态只在每个时钟周期结束后才会发生改变,因此,PDES 技术适用于处理器核的模拟。需要重点关注的两类保守同步机制是 barrier 同步和松弛同步 (slack synchronization),在多处理器模拟领域,保守的 PDES 方法是迄今为止研究最多的方法。

量子模拟

总体来讲,CMP 并行模拟中最为常见的保守同步机制是 barrier 同步。

最保守的方法是在目标处理器核每个周期结束后对模拟线程进行同步,这种方法与单线程逐拍 (Cycle-by-Cycle, CC) 模拟一致。在这种方法中,每个目标处理器核模拟的时间单位是一个周期,且在进入下个周期前,每个目标核的模拟线程必须等待其他所有核模拟线程完成本周期的模拟。图 9-5 显示了 4 个目标核映射到 4 个宿主机核上的逐拍并行 CMP 模拟的时序过程。所有的模拟线程都在每个模拟周期结束后进行同步,例如,在模拟时钟 1 时,P4 是目标核在第一个周期第一个完成模拟的线程,而 P1 是最后一个,因此所有的模拟线程在进入下一个周期前,必须等待 P1 的模拟完成。在逐拍模拟中,每个模拟线程都执行其目标核的一个周期,然后需要与其他线程进行同步,由于总有 3 个宿主机核在结束其模拟目标周期后处于空闲状态,因此这种频繁的同步会浪费并行性。

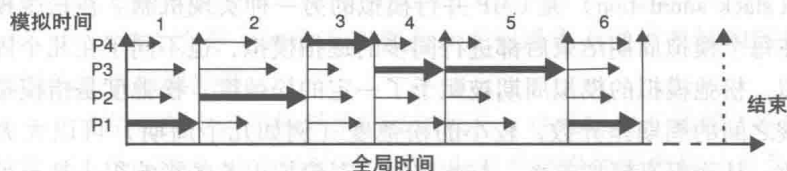


图 9-5 逐拍模拟

其实我们也可以在几个周期后对模拟线程进行同步，而不是每个周期都同步。模拟过程被分成由几个模拟周期组成的时间间隔，这些时间间隔由同步 barrier 进行分隔，每两个同步 barrier 之间的时间间隔称为一个量子，这种模拟方法通常也称为量子模拟。在一个量子期间，所有的模拟线程在达到同步点之前都可以独立推进执行，当模拟进入下一个时间间隔之前，在当前时间间隔内所触发的所有全局事件必须对所有模拟处理器核可见，这样它们产生的效果才能在下一个时间间隔中进行模拟。图 9-6 显示了基于量子的 CMP 模拟时序过程，图中所有处理器核模拟线程必须每 3 个周期同步一次，即该例中的量子时长为 3 个周期。图中当 P4 完成第一个周期的模拟时，不同于逐拍模拟，它无需等待 P1 模拟的完成，而是继续进行第 2 和第 3 周期的模拟。由于目标核中 P2 完成 3 个周期模拟所需的时间最长，因此其他线程在完成 3 个周期模拟后需要等待 P2。

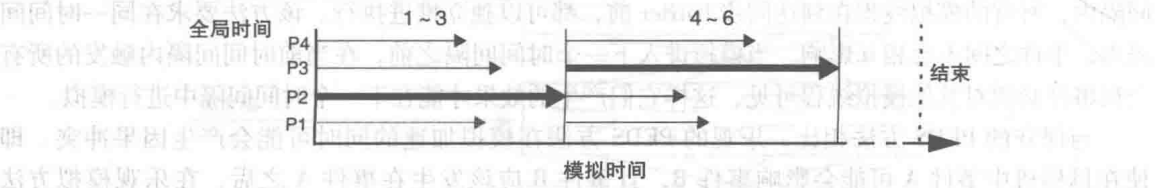


图 9-6 量子模拟

逐拍模拟和量子模拟的区别在于何时以及如何使得来自处理器核线程的需求成为全局可见的。在逐拍模拟中，处理器核线程在每个周期结束后进行同步，这种紧密同步保证了所有需求的影响效果可以立即成为全局可见的，从而使得模拟系统在每个周期结束后保持一致。另一方面，在量子模拟中，直到每个量子结束后请求才是全局可见的，在每个处理器核线程耗尽量子时间后，模拟器对共享资源进行更新，从而使得目标系统对下一个量子中的所有处理器核线程来说是状态一致的。

由图 9-6 可以看出，量子模拟明显比逐拍模拟效率更高，因为与逐拍模拟相比，量子模拟中的全局同步点减少了三分之二。一般来说，同步点越少，模拟效率越高。然而，由于线程到达同步点所需的时间是不同的，因此模拟速度受限于每个量子周期中速度最慢的线程。

量子模拟的准确性取决于量子的大小。当量子大小不超过将一个目标处理器核产生的事件传播到可以影响另一个处理器核模拟（如通信、同步或资源冲突等）的位置所需的最小延迟时，可以认为量子模拟同逐拍模拟一样准确，这个最小延迟称为临界延迟。在一个特殊的模拟系统中，识别临界延迟很可能非常困难，需要小心处理。例如，CMP 的线程经常因为共享资源（如处理器核和 L2 bank 之间的互连）而发生冲突，这种冲突可能只有一个周期的延迟，此时量子将设置为一个时钟周期，然而，人们也常常会忽略这种底层交互的影响，以便得到一个更为合理的量子大小。

使用 barrier 同步并行模拟的一个著名例子是 WWT II（Wisconsin Wind Tunnel II）模拟器，它是一个直接执行的离散事件模拟器，可以在共享内存多处理器或工作站网络上并行执行。

松弛模拟

松弛模拟（slack simulation）是 CMP 并行模拟的另一种实现机制。在松弛模拟中，模拟处理器核不同于在每个模拟周期结束后都进行同步的逐拍模拟，也不同于在几个固定周期后进行同步的量子模拟，松弛模拟的模拟周期被赋予了一定的松弛度。松弛度是指模拟过程中任意两个目标处理器核之间的周期差异数，较小的松弛度（例如几个周期）可以大大减少模拟线程之间的同步次数，从而提高模拟效率，与此同时，对模拟误差的影响很小甚至可以忽略。

图 9-7 给出了有界松弛模拟方法，线程间的最大松弛度类似于量子模拟进行限定，但是模

拟线程并不在 barrier 点处进行定期同步, 最大松弛度会限制所有模拟线程必须在一个周期窗口范围内, 该窗口的大小取决于最大松弛度, 只要最快的目标处理器核与最慢的目标处理器核之间的差距在一个松弛度范围之内, 那么目标处理器核就可以继续执行。在一个最大松弛是 S 个周期的窗口, 当最快的目标核比最慢的目标核快 S 个周期时, 最快的目标模拟器就需要等待最慢线程执行推进, 当最慢线程推进一个时钟周期时, 最快与最慢线程之间的差距缩小到 $S - 1$ 个周期, 此时, 最快的线程可以立即开始执行它的下一个模拟周期。

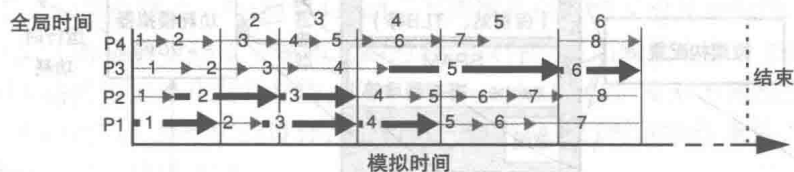


图 9-7 松弛模拟

模拟线程的局部时间是其目标处理器核的模拟周期计数, 所有模拟线程中最小的局部时间就是全局时间。参考图 9-7, 每个窗口的左边界总是与全局时间 (T_g) 相等, 当最大松弛为 S 个周期时, 每个模拟窗口的右边界是 $T_g + S$, 这是所有模拟线程的最大局部时间, 每当全局时间增加时, 窗口都会随之滑动, 而这种滑动出现在最慢的线程向前推进时。由于全局时间总是增加的, 因此窗口总是向右边滑动, 即朝着模拟结束的方向。在窗口内的所有线程都可以独立运行, 只有当局部时间等于 $T_g + S$ 时, 模拟线程才会被阻塞。在图 9-7 中, 松弛度为 2 个周期, 箭头指向全局时间的变化方向, 当全局周期为 3 时, P1 和 P2 同时结束模拟, 因此, 下一次模拟从全局周期 4 开始。

松弛模拟和量子模拟之间存在着显著的差异: 在量子模拟中, 只有当所有线程的局部时间等于它们的最大局部时间时, 全局时间才进行更新; 与之不同的是, 在有界松弛模拟中, 只要最小局部时间取得进展, 全局时间就进行更新, 尽管有界松弛模拟中的线程之间仍存在一些同步, 但相对于量子模拟来说, 这种同步要宽松得多。理想情况下, 在整个模拟期间, 可能没有任何线程需要阻塞以等待其他线程的执行, 这等同于假设所有线程始终运行在由松弛边界定义的滑动窗口内, 因此任何线程都不会到达它的上界。

9.5 功耗和热量模拟

对芯片设计师来说, 虽然性能模拟很有吸引力, 但与此同时, 也亟需一些其他的准确量化设计指标, 例如, 功耗和温度等。事实上, 提高性能的设计方案往往是以更高的功耗为代价的, 除功耗外, 热量约束也是设计时需要重点考虑的一项指标, 由于供电成本和制冷方案成本的不断提高, 这些设计指标变得同性能一样重要。因此, 在模拟设计时, 必须全面权衡功耗、热量以及性能之间的准确模拟, 这样才能做出最好的设计决策。

Wattch 是一个著名的体系结构级功耗模拟工具, 图 9-8 给出了其结构框图。像 Wattch 这样的体系结构功耗模拟器, 都是依赖于众所周知的动态功耗方程 $P = \alpha C V_{dd}^2 f$ 来建模。在这个方程中, α 为活动因子, 它主要统计组件通电时周期所占的比例; C 为组件电容; V_{dd} 为电源电压; f 为时钟频率。 V_{dd} 和 f 这两个参数与制造工艺相关, 它们通常不受体系结构的控制, 因此, 这些参数通常作为输入提供给模拟器。

测量处理器中每个组件的电容是极具挑战性的任务, 这是因为 C 依赖于制造工艺和系统微架构模块的电路实现。为了简化对 C 的计算, Wattch 将系统微架构模块分为 5 类: (1) SRAM 为主的阵列结构, 如 cache、寄存器堆和取指队列; (2) 内容可寻址存储 (CAM) 结构, 如保

留站和 TLB; (3) 线结构, 如总线; (4) 组合逻辑模块, 如功能单元; (5) 时钟分布的基础架构。

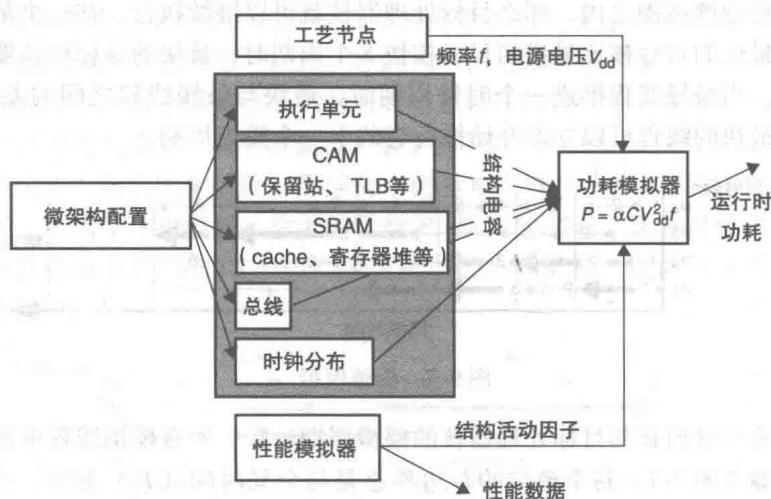


图 9-8 Wattch 模拟器

对 SRAM 为主的结构来说, 电容主要取决于线宽、读/写端口数以及 SRAM 阵列中的 bank 数。在 CAM 中, 电容主要取决于 CAM 中标识位 (tag) 和匹配位的数目, 它直接正比于 CAM 结构的大小。总线的电容正比于总线的长度以及总线上中继器的个数。系统微架构配置参数不包括总线长度, 它们取决于具体实现, 因此, 在系统微架构级别, 只能获取连接两个组件总线的近似长度, 或者, 用户也可以基于位置和路由工具来实现所提出的设计方案, 继而完成总线长度的测量。功能单元, 如加法器、逻辑单元、乘法器或浮点单元都是在门级设计的, 从而可以估计它们的功耗, 然后对这些设计进行布局布线, 从而得到这些功能单元的完整分布和面积。此外, 还需要对门级设计进行分析, 从而确定每个门的有效负载 (active load) (例如, 从当前门接收输入的门的数量)。对于那些需要驱动较大有效负载的门, 可以扩大门的尺寸以减小这些活跃门的传输延迟瓶颈, 在 ALU 中, 门尺寸增加会增大面积开销, 但同时可以减小 ALU 内门延迟的不均衡, 进而减少整个电路的功耗开销。所有经过均衡考虑和功耗优化的 ALU 设计的电容都可以在设计阶段完成后进行测量, 需要注意的是, 这种方法测量出来的电容是和制造工艺相关的, 当使用新的制造工艺来实现 ALU 时, 可以使用第 2 章中讲述的扩展法则来计算 ALU 的电容。最后, 时钟分布网络的电容取决于时钟驱动器和金属导线。

对每一类部件都可以通过参数化结构尺寸来计算电容。例如, 计算 SRAM 的电容时, 可以使用以位线数 N 和字线数 M 作为参数的函数; 而计算直接映射 cache 结构的电容时, 设计人员可以简单地使用 cache 行宽度 N 和 cache 行数量 M 作为参数。每个系统微架构模块都可以大致被划分到这 5 个类别中的一个, 并且电容都可以通过使用结构尺寸的参数值来进行测量。

最后, 在计算动态功耗时, 需要知道活动因子 α , 不同的系统微架构的活动因子可以通过性能模拟得到, 而活动因子依赖于模拟器上执行的基准测试程序。例如, 通过体系结构模拟器来测量 cache 行访问次数是十分容易的, 在这个过程中, 对每个感兴趣的组件都需要设置一个相关的计数器, 每进行一次访问, 相应的计数器就加 1。值得一提的是, 所维护的计数器的粒度取决于所需的详细程度, 如果设计人员想要模拟每个 cache 行的功耗, 则对每个 cache 行来说, 都需要维护一个计数器; 另一方面, 如果功耗测量是针对整个 cache 的, 则只需要维护一个计数器, 该计数器需要对任何 cache 行的所有访问进行计数。

虽然电路中的动态功耗占据了大部分的功耗开销,但漏电功耗也是研究人员十分关注的一个问题,因此,在 HotLeakage 工具中就实现了对漏电功耗的模拟。漏电功耗正比于电源电压 (V_{dd})、电路中的晶体管数 (N),每个晶体管的漏电流 (I_{leak}) 以及一个特定的称为 K_{design} 的参数。需要注意的是,对于任何设计, V_{dd} 和 N 都很容易确定,对于那些不使用任何 DVFS 和动态重配置的电路来说(比如,不支持诸如关闭一个未使用的 ALU 这样的技术), V_{dd} 和 N 的值在设计时就被固化了。然而,对于那些使用类似于 DVFS 技术的设计,在每次 DVFS 调节后, V_{dd} 都会发生变化,如果电路的一些组件可以动态关闭或开启,则 N 也是可变的。

参数 I_{leak} 和 K_{design} 都依赖于具体实现和电路的操作条件,例如阈值电压、电源电压和温度。HotLeakage 使用电路模拟工具开发的方法,如 Berkeley Spice 模拟,来对不同操作条件下的这些参数进行计数,HotLeakage 对给定的设计进行模拟,并测量运行时的操作条件,主要是温度和电压,用以计算上述两个影响参数。

HotSpot 是一个用于温度模拟的工具,它将设计布局作为输入,设计布局指定各种电路模块以及模块间的连接关系。它可以指定不同的模块粒度,例如,可以指定流水阶段这样的粗粒度模块,或指定如 ALU 的细粒度模块,或一组重排序缓冲项等。然后,HotSpot 将每个模块转换成一个简单的集中 RC 模型。处理器设计中,每个模块都与其相邻的模块连接,模块被建模为电阻器,因此模块间的连接被用来创建整个设计的 RC 网络模型。HotSpot 也对那些堆叠在处理器之上的,用以降低温度的散热器和散热板进行建模。要获得精确的温度估计,如何解释这些组件对整体温度的影响是至关重要的,散热器和散热板的影响也建模成集中 RC 模型。处理器设计的每个模块都与其垂直层(散热器和散热板)存在连接,该连接被建模为热电阻,整个 RC 模型被称为热阻简化模型(Compact Thermal Model, CTM)。接下来在处理器设计的模拟执行中可以测量通过每个模块的电流,继而 CTM 使用电流信息来确定下层 RC 网络的温度。

9.6 工作负载采样

由于计算机体系结构设计人员还严重依赖于模拟的方法来评估未来处理器的性能,因此模拟加速就显得至关重要。正如前面所讨论的,并行是加速模拟的一个重要方法,在并行模拟中,目标系统不同的组件(如目标 CMP 中的处理器核)都是通过并行的线程进行模拟的。和并行模拟正交的另外一种模拟加速方法是工作负载采样,在这种方法中,模拟线程只对目标机上的部分工作负载执行过程进行模拟,整体性能指标由运行样本的测量值推测得出,为了使结果具有可信度,样本的模拟运行必须可以代表整个工作负载的执行。

在最简单、最原始的工作负载采样形式中,我们根据设计人员所提供的模拟时间限制,只模拟固定数量的指令。比如,假设模拟器每分钟可以模拟目标系统结构下的 100 万条指令,那么设计人员就可以在大约 17 小时内模拟 10 亿条指令。如果整个基准测试程序有 100 亿条指令,那么一个简单的策略是只模拟基准测试程序的前 10 亿条指令,因为只采样所有指令的 10%,模拟时间加速了近 10 倍。如果在样本运行期间,发生了 1000 万次的 cache 失效,则设计人员可以推测出,整个基准测试程序在其执行过程中将会发生 1 亿次 cache 失效,这是假设在采样窗口内外,cache 失效率始终保持一致,换句话说,样本运行时的 cache 失效率可以代表整个基准测试程序的 cache 失效率,当然,这只是一种理想状态下的情况。

大多数基准测试程序将它们最初的执行时间花费在读取输入并转换成便于计算的格式这一过程上。例如,读取一组数值来填充内存中的一个矩阵,因此,在上面提到的这种原始的采样方法中,如果目标是测量 CPU 和内存系统的性能,最好是要跳过程序的初始执行阶段,因此,对上面采样方法的一个简单改进是跳过初始设置阶段。根据设计人员对基准测试程序的了解,每个基准测试程序需要跳过的指令数都可能不同。注意这种做法存在一个明显的假设,即在跳

过指令后可以重建系统状态,换句话说,如果设计人员选择跳过前1亿条指令,在跳过这些指令后,应该可以生成对应的寄存器状态和内存状态。大多数模拟器都支持两种模式的操作:功能模拟模式和详细微架构模拟模式。功能模拟模式只在每条指令执行后,跟踪记录系统架构状态的转变,它并不模拟微架构状态,因此,功能模拟模式比详细模拟模式运行速度快得多。在详细模拟模式中,整个系统微架构的状态都被模拟,包括跟踪指令依赖关系、模拟指令乱序执行过程、模拟 cache 和分支预测等。为了跳过基准测试程序的最初阶段,可以先开启快速的功能模拟模式,然后切换到速度较慢的详细微架构模拟模式下对样本进行模拟。

上述原始采样方法存在的一个问题:样本运行结果也许并不能够精确反映整个基准测试程序的行为,这是因为样本的选择存在随意性,一些与度量标准相关的、研究人员最感兴趣的执行事件可能发生在采样窗口外,从而导致不准确的结论。

9.6.1 基于采样的微架构模拟

相比于简单的随意采样方法,样本的质量和模拟结果的准确性可以通过基于推论统计学的系统随机采样得以改善。除了计算机体系结构领域外,随机采样还广泛应用于其他诸多领域,这是因为随机采样的偏差小,由采样引起的误差可以通过统计参数(如置信级别和误差范围)进行量化。系统随机采样是一种选择样本的方法,在这种方法中,所有被选择的样本之间是等距的,也称为均匀采样。换句话说,采样间隔在整个程序执行过程中是等距的,与任意选择一个连续的样本相比,这种方法可以更大程度地捕获整个基准测试程序的行为。例如,从一个运行100亿条指令的程序中选择1000个样本,每个样本由100万条指令组成,那么将在每1000万条指令后抽取一个样本。基于采样的微架构模拟器(如SMARTS)的目标是:在程序整个执行过程中来选择样本的数量和每个样本的规模,以满足选择度量标准所需的置信区间。

SMARTS 先从功能模式开始,到达第一个采样点后,切换到详细模拟模式对样本区间进行模拟。样本区间模拟完后,在到达下一个采样模拟点之前,它将再次切换回功能模式。从已有的经验来看,SMARTS 建议在每个采样窗口使用1000条指令。模拟要达到的置信区间可以根据所关注指标的方差进行估计,并通过对采样速率进行调节来达到给定的置信区间。比如,假定CPI是关注的指标,SMARTS通过执行第一个1000条指令来测量采样窗口中的CPI方差,据此决定需要选取多少个样本来达到用户指定的置信区间。

微架构模拟的一个问题是,处理器中的一些结构比较庞大复杂,并且这些结构状态的保持对我们所关注的指标有显著影响,因此,当使用均匀采样抽取只有1000条指令的采样窗口时,如果在每个采样窗口开始时,这些微架构状态没有被正确地设置,那么测量出来的性能指标往往是不准确的。例如,在采样窗口期间,分支预测的准确性会影响CPI,而分支预测的准确性取决于在采样窗口开始前预测结构如何捕获分支历史,由于分支预测器有大量的状态,在只有1000条指令的采样窗口中,期望预测器状态被完整填充这是不现实的,实际上,这些大型结构需要再多几个数量级的指令才能够完成状态的准确填充。和分支预测结构类似,cache结构也有类似的问题。不过,处理器中的小型缓冲区(如重排序缓冲区)可能不需要预先填充,因为这些结构非常小,即使在采样窗口中对其状态进行填充也不会引入太大的误差。因此,系统性的采样方法不能完全依赖于功能模拟和详细模拟之间的切换,那些包含大量状态的系统微架构缓冲区必须在每个样本前的预热窗口中进行模拟,在预热模式下,小型的微架构缓冲区不需要进行模拟,从而提高了模拟速度。

预热的概念为采样方法带来了一个新的参数,即预热窗口长度。理想情况下,预热窗口长度取决于每个具体的微架构结构的大小。对分支预测器来说,其预热窗口的大小可能是上万甚至更多的指令序列;对cache来说,它包含更多的微架构状态,预热可能需要上百万条指令;

对较大的最后一级 cache 来说,甚至可能有必要执行整个基准测试程序来预热 cache。设计人员必须对准确性和模拟速度进行权衡来选取合适的预热参数。SMARTS 建议采用两级预热,先用 50 万条指令对 cache 进行预热,紧接着使用 4000 条指令对其他微架构部件进行详细预热,预热完成后进入 1000 条指令对应的采样窗口,在采样窗口中完成对性能指标的实际测量。

9.6.2 SimPoint

另一种用于选取采样窗口的方法是基于程序特征的。在一个典型的计算机体系结构模拟中,很多情况下都是以相同的输入设置、不同的目标系统结构配置反复地运行大量的基准测试程序。每个基准测试程序的执行期间可以看成是执行一组基本块,且基本块的执行模式往往是频繁重复的。因此,可以将整个程序的执行归纳为几个阶段,在执行过程中,各阶段重复执行的次数是不同的。基于各阶段重复执行的次数,可以将整个程序的执行进行聚类,并识别出执行过程中占主导地位的关键阶段,可以选取单个的关键阶段或一组最主要的关键阶段作为样本。同其他采样方法一样,SimPoint 也先通过功能模拟的方式快速推进到关键阶段,然后切换到详细模拟模式对关键阶段进行模拟。

SimPoint 是基于阶段分析来加速模拟的最流行工具,阶段的识别和聚类是通过使用基本块向量来实现的。在 SimPoint 中,程序的每个基本块有一个独特的标识符或基本块号,程序执行被分割为 1 亿条指令的窗口,每个窗口中基本块被执行的次数被记录在称为基本块向量(BBV)的结构中,程序中的每个基本块都在 BBV 中有对应项。BBV 并不是精确地捕获每个基本块的执行顺序,而是统计这 1 亿条指令窗口中每个基本块总的执行次数。第一步是识别 BBV 之间的相似之处,在极端情况下,如果在两个对应窗口中,每个基本块的执行计数都是相同的,则相应的两个 BBV 是相同的。然而,事实上很少有多于一个 BBV 的计数恰好相同,因此两个向量之间的相似度可以使用欧氏距离或曼哈顿距离进行度量,一旦 BBV 之间的相似度被计算出来,下一步就是将每个 BBV 分类到 k 个类簇中,这可以通过著名的 k -means 聚类方法来实现。

k -means 聚类方法使用一个两阶段的迭代过程来实现 BBV 的聚类:第一步,随机选取 k 个点作为每个聚类的中心,然后计算每个 BBV 到不同聚类中心的距离,并将该 BBV 放置到距离最近的那个聚类中;第二步,根据第一步的划分计算新的聚类中心,再重新计算每个 BBV 到新的聚类中心的距离,一些 BBV 可能会重新分配到不同的聚类中。这是一个迭代的过程,当没有 BBV 重新分配时,迭代停止。

当所有 BBV 聚类到 k 个类簇后,可以在每个类簇中选取一个 BBV 来代表整个类簇,因此可以只模拟每个类簇中选取的 BBV 窗口(每个 BBV 窗口有 1 亿条指令)来代表整个程序的执行,被选中的窗口或阶段称为 SimPoint。在极端情况下,可以选择到程序中所有 BBV 的欧氏距离或曼哈顿距离最近的一个 BBV,这种选择导致只需要进行单个 SimPoint 的模拟。不同于 SMARTS,SimPoint 中的每个阶段都执行 1 亿条指令,因此,即使阶段模拟开始时的微架构缓冲区都是空的,这些缓冲区结构也可以在阶段开始后很快达到稳定状态,因此也没有必要像 SMARTS 方法中那样对系统微架构部件进行预热。

9.7 工作负载特征刻画

工作负载特征刻画的目的是多方面的:首先,它帮助研究人员理解运行在当前或未来机器上的一组程序的性能瓶颈;其次,它为开发那些用来描述工作负载的综合基准测试程序提供关键信息;最后一点但同样重要的是,当工作负载增加时,基于工作负载特征刻画可以推测出其在现有或未来机器上的行为。在最后那种情况下,工作负载特征刻画是另一种加速模拟的方式,这种加速是通过减少工作负载的大小和预测更大工作负载的行为来实现的。

9.7.1 理解性能瓶颈

工作负载特征刻画的核心目标是,描述工作负载是如何与底层目标架构进行交互的。工作负载特征刻画可以使设计人员充分理解有哪些系统微架构特征可以提高或阻碍工作负载的性能,如果工作负载足够重要,那么在某个微架构上的工作负载特征刻画可以帮助设计人员对未来的微架构作出决策。例如,工作负载在执行过程中遇到过多的 TLB 失效,则未来的设计可能会增加 TLB 的容量,从而缓解由于 TLB 失效而引发的性能损失。除此之外,工作负载特征刻画也能够帮助软件开发人员理解如何重写代码,以避免一些潜在的性能损失。工作负载特征刻画可以通过在当前的真实物理系统上设置和运行工作负载来完成,当工作负载运行在当前系统上,可以通过硬件计数器来收集指令执行信息和微架构行为统计数据。当然,工作负载特征刻画也可以在模拟的未来系统上得以实现,并且模拟的方法为工作负载和目标系统之间的微架构交互提供了更多的可见性。

VTune 是现有的性能分析工具代表,它是商业软件,主要针对 Intel 架构上的软件进行性能分析,它可以分析运行在本地硬件上(包括 OS 在内)的任何程序,也包括运行在多处理器上的多线程程序,并且这些程序都不需要重新编译源代码。这种工具在分析复杂多线程服务器程序时特别有用,因为多线程服务器程序由于缺乏源代码,既无法做代码插桩,也无法重编译。当程序在物理机器上执行时,这些工具可以通过嵌入在处理器中的事件计数器来监测大量性能或代码的执行属性。例如,VTune 可以在给定的时钟周期内,收集程序计数器的取指次数、退出指令的条数、cache 失效次数、分支预测错误次数、总线访问次数,以及总线冲突次数等。VTune 定期中断执行,例如每百万条指令中断一次(指令个数可以通过测量退出指令得到),然后它在每个中断点处,对指令计数器和一些事件计数器的值(如时钟周期数或指令数)进行记录。在程序执行结束时,VTune 将这些对样本数据点的跟踪记录用于工作负载特征刻画。在最简单的情况下,VTune 可以估计出执行工作负载过程中不同类型指令的相对出现频率;而更加细致全面的特征刻画则可以区分出不同的分支指令类型,并给出不同分支指令下的分支预测准确性。

9.7.2 合成基准测试程序

工作负载特征刻画还有助于合成新的基准测试程序。一个简单的例子是收集在复杂工作负载上执行的指令混合信息,这一信息可以用来创建同初始工作负载具有相似指令特征的合成基准测试程序。这个合成基准测试程序不需要初始工作负载那样的复杂设置,就可以用来模拟未来的目标设计。当工作负载的设置难度较大时,这种方法就特别有用,比如,针对 TPC-C 这样的具有复杂工作负载的事务处理基准测试程序。需要注意的是,具有相似指令混合特征的合成基准测试程序可能无法展现服务器工作负载中的复杂系统级交互,因此,它的使用场合可能比较有限,而且通过合成基准测试程序所得结果的可信度也经常受到质疑。

9.7.3 预测工作负载行为

工作负载特征刻画的另一个重要应用是,当工作负载规模增大时,或工作负载迁移到未来不同的目标系统上时,工作负载特征刻画可以用来理解工作负载的不同行为。通常,用于产品环境下的商业工作负载对硬件和软件的要求极高,这种情况下的配置被称为扩展设置(scaled setup),系统供应商往往通过扩展设置来展示系统的最佳性能,比如,展示全世界最快的事务处理吞吐能力。这种设置需要使用大量的磁盘、PB 级别的内存,以及成千上万的处理器,通过每个处理器的性能计数器来将其调节到最佳性能。这种设置下的资源需求太大,无法通过模

拟环境实现,因此,通过这种设置来探索新的系统微架构的想法是不切实际的。事实上,对于很多的商业工作负载,当其运行在全工作模式下时,研究人员和设计人员是无法对其工作负载行为进行分析的。不同于 CPU 密集的基准测试组件(如 SPEC),商业工作负载(如 TPC-C)通常难以配置,这有多方面的原因:首先,商业工作负载需要仔细调节大量的配置参数,例如,在 TPC-C 的设置过程中,为各种数据库处理所分配的内存数量,或者磁盘数据分区方法,都可能会对整体性能产生明显影响;其次,商业工作负载中的源代码通常无法获取,这进一步增加了设置的复杂性;最后,过高的软硬件成本也使得扩展设置难以实现。

研究人员通常将这些复杂的工作负载限定在一个较小的设置中,称为缓存设置。缓存设置通过减少各方面所需的资源以获得更低的复杂度,通过限制负载只在少数几个处理器上运行,并限制内存或 I/O 需求,可以减少硬件的成本开销。现实中,随着工作负载的减少,硬件资源的需求也显著降低。通过缓存设置,原来需要运行一个包含 100 万用户的数据库,现在可能只需要运行包含 10 万个用户的数据库。虽然缓存设置仍然还是比较困难,但由于它只用了少量的系统存储和磁盘,因此配置已经相对容易很多。和实际产品环境相比,在研究或设计时期所采用的工作负载大小和典型 TPC-C 设置所对应的资源需求可能有多达 3 个数量级的差距。

工作负载特征刻画可以弥补缓存设置和扩展设置之间的差距,特征刻画可以对缓存设置下的运行结果进行分析,并预测工作负载在扩展设置下的潜在性能。这种特征刻画将有助于我们理解基于模拟的设计决定在真实产品环境下将表现出什么样的性能。因此,工作负载特征刻画的首要目标就是理解在缓存设置和扩展设置下系统行为的差异,并发掘能够刻画商业工作负载在不同配置下性能的潜在关系。

习题

9.1 我们使用主流模拟器来完成一些设计评价研究。SimpleScalar 是一组模拟器套件,它可以在不同细节层次上对机器进行模拟(相关资料见 <http://www.simplescalar.com>): sim-cache 是一个简单的 cache 模拟器,当输入一系列 cache 设计指标时,它可以测量 cache 的命中和未命中的次数;sim-outorder 是最详细的模拟器,它可以对具有动态调度、分支预测、推测执行以及 cache 等配置的超标量处理器进行逐拍模拟。下面,我们将使用 SimpleScalar 专门为教学而提供的基准测试程序(详细资料见 <http://www.eecs.umich.edu/~taustin/eecs573Vpublic/instructprogs.tar.gz>)。根据可用的计算资源,你可以运行该基准套件中任意数量的基准测试程序。

(a) 使用 sim-cache 来模拟以下 4 种 cache 配置方案。配置一: L1 指令和数据 cache 独立,各为 4KB,采用直接映射,cache 块大小为 32B;配置二: L1 指令和数据 cache 独立,各为 4KB,采用二路组相联映射,cache 块大小为 32B;配置三: L1 指令和数据 cache 独立,各为 16KB,采用直接映射,cache 块大小为 32B;配置四: L1 指令和数据 cache 独立,各为 16KB,采用四路组相联映射,cache 块大小为 32B。请分析 L1 cache 相联度对 cache 失效率的影响,并分析 L1 cache 大小和相联度是如何相互影响 cache 失效率的。

(b) 使用工具 Cacti(它可以对任意数据存储结构的访问时间和功耗进行评估,例如 ROB 和 cache 结构等),Cacti 的输入是 cache 大小、相联度以及生产工艺。读者可以自己下载 Cacti,或者使用网页版的功耗/面积评估器(<http://www.hpl.hp.com/research/cacti>)。使用 Cacti 分别计算问题(a)中 4 种 cache 配置方案下的延迟和功耗,使用 130nm 工艺,cache 配置为 4 个端口和一个 bank。假设功耗是最重要的设计因素,且已知 L1 失效率每提高 1%,相应的处理器功耗也提高 1%。功耗的提高是由于 cache 失效时,需要通过一个复杂的 cache 失效处理逻辑进行处理,且必须到下一级存储层次中读取失效的数据。请对比 Cacti 测量得到的 4 种 cache 配置下的功耗值,再将其与 cache 失效率变化引起的功耗值进行对比,选出哪种 cache 设计最有利于降低整体功耗。

(c) 使用 sim-outorder 模拟器来研究乱序处理器的设计空间。首先, 使用下面的参数来创建一个基线配置文件, 指定好一些关键结构的大小和延迟, RUU 和 cache 的延迟在这里没有给出, 我们需要通过 Cacti 来获取这些延迟的估计值。

考虑一个基线处理器具有如下的配置参数 (未指定的参数都使用默认值):

- 4 项取指队列。
- 两级 PAp 分支预测器, 每项分支历史占 4bit。
- 128 项历史表。
- 2048 项预测表。
- 512 项、128 组、四路组相联的分支目标缓冲区 (BTB)。
- 8 项返回地址栈 (RAS)。
- 每周期最多发射 4 条指令。
- 每周期最多译码 4 条指令。
- 每周期最多提交 4 条指令。
- 64 项寄存器更新单元 (RUU)。
- 32 项 load/store 队列。
- 3 个整形 ALU。
- 1 个整形乘法器/除法器单元。
- 1 个浮点 ALU。
- 1 个浮点乘法器/除法器单元。
- 32B cache 块。
- 16KB 四路组相联, L1 指令 cache, LRU 替换算法。
- 16KB 四路组相联, L1 数据 cache, LRU 替换算法。
- 512KB 八路组相联, 指令和数据共用的 L2 cache, LRU 替换算法。
- 4 个内存端口。
- 首次主存访问延迟为 100 个时钟周期, 后续访问延迟为 30 个时钟周期。
- 100 个时钟周期的 TLB 延迟。

使用 Cacti 来评估 L1 和 L2 cache 的访问延迟, 以 RUU 的访问延迟为单位。

步骤 1: 使用 Cacti 来计算 RUU 的访问时间, 假设 RUU 的更新在关键路径上, 并且处理器的运行频率取决于该延迟。

步骤 2: 使用 Cacti 来估计 cache 的访问时间, 需将该访问时间转化为处理器的时钟周期数。例如, 获得 RUU 的访问时间为 1ns, L1 数据 cache (DL1) 的访问时间为 10ns, 则 DL1 的延迟为 10 个时钟周期, 使用 130nm 的生产工艺, 设置端口数为 4, bank 数为 1。

步骤 3: 使用 Cacti 估计的 RUU 和 cache 延迟来完成基线配置文件。

步骤 4: 完成配置文件后, 对基线处理器的配置进行模拟, 模拟基线配置过程中需要使用本题开始时给出的 SimpleScalar 网站上提供的两个基准测试程序。

(d) 探索机器宽度如何影响超标量动态调度处理器的性能, 在配置文件中改变机器的宽度 (取指/译码/发射/提交的宽度)。为了探索设计空间, 需要使用以下的发射宽度: 每个时钟周期发射 1、2、4、8 条指令, 需要注意的是, 即使是这个简单的设计空间探索过程中, 每改变一个设计参数, 也需要针对每个基准测试程序模拟运行 4 次, 从这个例子我们也可以对设计空间探索所需的计算复杂度有个直观的感受。

步骤 1: 生成结果, 并绘图分析以给出你对设计空间探索的理解。

步骤 2: 绘制出 MIPS 和机器宽度关系的曲线图。观察各种评估指标, 如各种缓冲区的占用 (取指队列, RUU, load/store 队列)、分支指令的动态出现频率、cache 命中率等, 并给出这些曲线的相应解释。要获得 MIPS 值, 你需要知道处理器的运行频率, 该频率可以通过 RUU 的访存时间来确定。

9.2 在 9.2.3 节中, 给出了基于 ATOM 的伪代码, 使用直接执行的方式完成 cache 模拟。在本题中, 需要设计一个基于 PIN 的直接执行工具来模拟未来的 cache 设计。首先, 需要下载 PIN 工具组件 (下载网址: <http://www.pintool.org/>), PIN 可以在任何运行 Windows、Linux 或 Mac 系统的 x86 笔记本/台式机上运行, 安装完成后, 阅读下载包中附带的插桩和使用指南。

(a) PIN 有一个用于 cache 模拟的简单例子, 称为 pinatrace, 其代码包含在下载组件中, 也可以在 PIN 网站上的示例介绍区域找到 (网址: <http://www.pintool.org/docs/41150/Pin/html/>)。pinatrace 提供了如何使用 PIN 来收集内存引用 trace 的基本知识, 每个 trace 记录的格式是〈内存指令 PC, 读/写类型, 存储访问的虚地址〉。开始 trace 收集之前, 前 1 亿次内存引用将被跳过, 然后将接下来的 1000 万个 trace 记录保存到指定的输出文件中。你可以运行所选择的任何程序, 如果有许可 (license), 可以在 www.spec.org 上下载一组用来对计算机体系结构评估的基准测试程序, 若无法下载, 也可以运行任何桌面程序, 例如 Microsoft Office 组件或 GCC。需要注意的是, 无论选择哪个程序运行, 确保你已经正确设定好终止 trace 收集的条件, 否则, 可能会产生大量的 trace 文件, 并对你的计算机文件系统造成破坏。

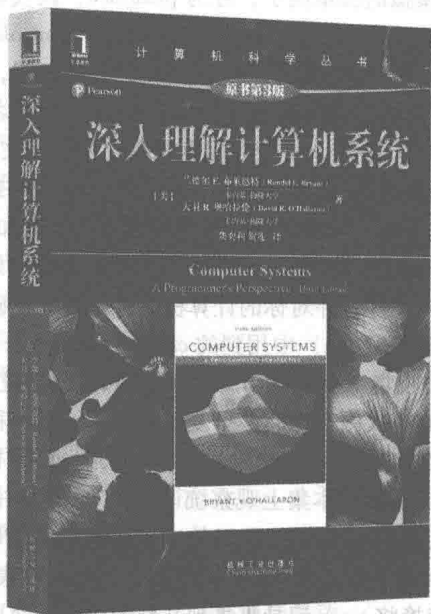
(b) 将 trace 输出文件输入到问题 (a) 中用到的 cache 模拟器中, 用 trace 文件重做上题。

(c) 使用 SMARTS 采样方法收集一个新的 trace, 重写 PIN 的插桩代码, 使之跳过前 1000 万条指令, 并收集接下来的 2000 次内存访问。重复这个过程, 直到程序终止或收集到的 trace 记录的数目超过 1000 万, 现在将这个新 trace 输出文件输入到 cache 模拟器, 重做问题 (a)。请分析在 SMARTS 样本和之前 trace 样本集上观察统计到的 cache 命中/未命中的变化情况。

(d) 现将 cache 模拟器直接集成到 PIN 工具组件中, 通过修改 PIN 中的 trace 收集代码, 直接将 load/store 指令中的访问地址输入到 cache 模拟器, 而不是收集大量的 trace 文件, 即使用 cache 模拟器的功能调用来替换将 trace 记录收集到文件的功能。使用这种方法, 不需要收集 trace 记录, 因此运行基准测试程序时不需要担心 trace 文件的大小, 分别使用问题 (a) 中的简单样本和问题 (c) 中 SMARTS 样本运行全部基准测试程序, 观察比较 cache 命中/未命中率的相关情况。

9.3 图 9-6 和图 9-7 的量子模拟和松弛模拟展示了松弛模拟如何在量子模拟的基础上进一步提高性能。假设量子是 3 个时钟周期, 而松弛度是 4 个时钟周期, 假定在一个 4 核 CMP 宿主机上模拟一个 4 核目标 CMP, 请问在什么情况下, 量子模拟和松弛模拟的加速效果是相同的?

推荐阅读



深入理解计算机系统（原书第3版）

作者：[美] 兰德尔 E. 布莱恩特 等 译者：龚奕利 等 书号：978-7-111-54493-7 定价：139.00元

理解计算机系统首选书目，10余万程序员的首选

卡内基-梅隆大学、北京大学、清华大学、上海交通大学等国内外众多知名高校选用指定教材
从程序员视角全面剖析的实现细节，使读者深刻理解程序的行为，将所有计算机系统的相关知识融会贯通
新版本全面基于X86-64位处理器

基于该教材的北大“计算机系统导论”课程实施已有五年，得到了学生的广泛赞誉，学生们通过这门课程的学习建立了完整的计算机系统的知识体系和整体知识框架，养成了良好的编程习惯并获得了编写高性能、可移植和健壮的程序的能力，奠定了后续学习操作系统、编译、计算机体系结构等专业课程的基础。北大的教学实践表明，这是一本值得推荐采用的好教材。本书第3版采用最新x86-64架构来贯穿各部分知识。我相信，该书的出版将有助于国内计算机系统教学的进一步改进，为培养从事系统级创新的计算机人才奠定很好的基础。

——梅宏 中国科学院院士/发展中国家科学院院士

以低年级开设“深入理解计算机系统”课程为基础，我先后在复旦大学和上海交通大学软件学院主导了激进的教学改革……现在我课题组的青年教师全部是首批经历此教学改革的学生。本科的扎实基础为他们从事系统软件的研究打下了良好的基础……师资力量补充又为推进更加激进的教学改革创造了条件。

——臧斌宇 上海交通大学软件学院院长

推荐阅读



计算机组成与设计：硬件/软件接口（原书第5版）

作者：[美] 戴维 A. 帕特森 等 ISBN: 978-7-111-50482-5 定价：99.00元

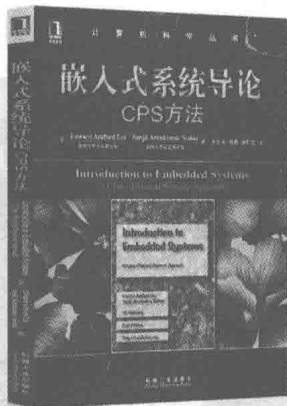
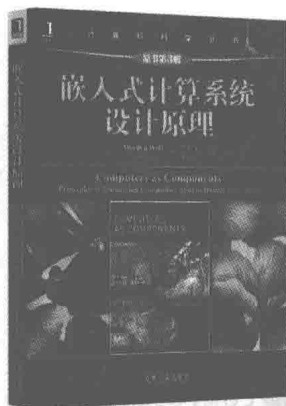
本书是计算机组成与设计的经典畅销教材，第5版经过全面更新，关注后PC时代发生在计算机体系结构领域的革命性变革——从单核处理器到多核微处理器，从串行到并行。本书特别关注移动计算和云计算，通过平板电脑、云体系结构以及ARM（移动计算设备）和x86（云计算）体系结构来探索和揭示这场技术变革。

计算机体系结构：量化研究方法（英文版·第5版）

作者：[美] John L. Hennessy 等 ISBN: 978-7-111-36458-0 定价：138.00元

本书系统地介绍了计算机系统的设计基础、指令集系统结构，流水线和指令集并行技术。层次化存储系统与存储设备。互连网络以及多处理器系统等重要内容。在这个最新版中，作者更新了单核处理器到多核处理器的历史发展过程的相关内容，同时依然使用他们广受好评的“量化研究方法”进行计算设计，并展示了多种可以实现并行，陛的技术，而这些技术可以看成是展现多处理器体系结构威力的关键！在介绍多处理器时，作者不但讲解了处理器的性能，还介绍了有关的设计要素，包括能力、可靠性、可用性和可信性。

推荐阅读



嵌入式计算系统设计原理（原书第3版）

作者：玛里琳·沃尔夫 译者：李仁发等 ISBN：978-7-111-44075-8 定价：69.00元

本书从组件技术的视角出发，讲述了嵌入式计算的基本原理和技术。全书每一章涵盖一个专题，包括与嵌入式系统设计相关的若干主要内容：指令系统、CPU、计算平台、程序设计与分析、进程和操作系统、系统设计技术以及多处理器和网络等。

本书内容丰富，文字通俗流畅，讲述深入浅出，配合了丰富的设计示例与编程示例，使得读者在进行理论学习的同时，能够较容易地联系实际，加深对嵌入式计算设计思想的理解，并获得先进的技术实践指导。

嵌入式系统导论：CPS方法

作者：Edward Ashford Lee 等 译者：李实英等 ISBN：978-7-111-36021-6 定价：55.00元

本书是一本关于CPS（信息物理系统）的著作。不同于大多数嵌入式系统的书籍着重于计算机技术在嵌入式系统中的应用，本书的重点是论述系统模型与系统实现的关系，以及软件和硬件与物理环境的相互作用。

全书从CPS的视角，围绕系统的建模、设计和分析三方面，深入浅出地介绍了设计和实现CPS的整体过程及各个阶段的细节。

软/硬件协同设计（原书第2版）

作者：帕特里克 R. 肖蒙 译者：王奕等 ISBN：978-7-111-52018-4 定价：89.00元

本书介绍如何将自定义硬件集成到软件的嵌入式系统设计中，阐释了如何解决计算机工程中的关键问题：嵌入式系统设计者如何平衡设计的灵活性和高效性。本书内容覆盖了软/硬件协同设计的4个主题：基础概念、自定义体系结构的设计空间、软/硬件接口和应用实例，此外还介绍了设计环境，以便帮助读者开展软/硬件协同设计实验。本书包括基于Xilinx和Altera现代FPGA环境的实验与示例，这使得本书的内容适用于很多使用这些工具的课程。

并行计算机组成与设计

Parallel Computer Organization and Design



书中没有晦涩抽象的技巧以及使人手足无措的大量数据，而是以完整且易于教学的方式组织成章，并且包含片上多处理器等紧跟工业发展前沿的内容，最后一章量化评估更是点睛之笔。

—— Mikko Lipasti, 威斯康星大学麦迪逊分校

这本书不仅可以帮助你清晰理解并行系统的原理，而且对于并行系统设计者来说也是不可多得的好书。

—— 陈云霁, 中国科学院计算技术研究所

并行体系结构是计算机系统获得高性能和高效率的关键，与此同时，并行编程困难和设计瓶颈也带来了重重挑战。这一领域的知识较为艰深，技术更新迅速，因此，教育界、学术界和企业界都在渴望一本综合性强、权威性高但又浅显易读的书，无疑，本书将是最佳选择。

本书特色

- **清晰简明，易于入门。**从处理器基础知识到内存结构再到片上多处理器架构，作者都能用易懂的描述和恰当的例子来阐释复杂概念；而且，书中并没有将传统单核体系结构的知识完全略去，因而初学者也可轻松上阵、步步迈进。
- **各章独立，内容完备。**无需通读全书便可快速学习某一章的主题，教学和自学皆便捷高效；在展开技术层面的深入讨论时，更加关注复杂性、功耗和可靠性等与时俱进的重要设计因素，这也是其他教科书鲜有触及的。
- **掌握工具，实践创新。**系统讲解了实际应用中的并行系统，且专门用一章的篇幅来介绍量化评估工具；在充分了解现有系统及其限制因素的基础上，特别鼓励读者思考并实现自己的设计，以达到实践和创新的高阶学习目标。

作者简介

米歇尔·杜波依斯 (Michel Dubois) 南加州大学电子工程系教授，研究方向为计算机体系结构、并行处理和计算机系统的性能评估。

穆拉里·安纳瓦拉姆 (Murali Annavaram) 南加州大学电子工程系副教授，研究方向为计算机体系结构、3D芯片堆叠和Mobiquitous计算。

佩尔·斯坦斯托姆 (Per Stenström) 查尔莫斯理工大学计算机工程系教授，瑞典皇家工程科学院会员。

CAMBRIDGE
UNIVERSITY PRESS
www.cambridge.org

投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn



上架指导: 计算机/计算机组成

ISBN 978-7-111-56223-8



9 787111 562238 >

定价: 99.00元

[General Information]

书名=并行计算机组成与设计=Parallel computer organization and design

作者=(美) 迈克尔·杜波依斯, (美), 穆拉里·安纳瓦拉姆, (瑞典) 佩尔·斯坦斯托姆著; 范东睿, 叶笑春, 王达译

页数=365

SS号=14182122

DX号=

出版日期=2017.04

出版社=机械工业出版社